

# Clock Domain Crossing

Amr Adel Mohammady

---

 /amradelm

## Part 1

- Sources of chip failure
- How metastability happens in a flip-flop
- Synchronous vs asynchronous clock domains
- CDC concerns
- Mean Time Between Failure (MTBF)

## Part 7

- Timing Constraint for CDC paths
- Applying False Paths
- Not Applying False Path
- Skew Constraint
- Max Delay Constraint

## Part 2

- Required pulse width
- The issue of varying settling time
- CDC Rules:
  - Convergence/Glitches
  - Multi-Clock fan-in
  - Divergence
  - Reconvergence

## Part 3

- Data duplication issue
- Pulse generator
- Edge detectors

## Part 4

- Gray coding
- Sending counter values across different clock domains

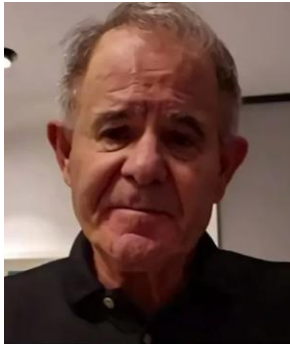
## Part 5

- Multi-bit Data CDC
- MUX Recirculation Scheme
- Handshake Protocol

## Part 6

- Circular FIFO
- Full And Empty Conditions
- Binary to Gray Encoder And Decoder
- CDC FIFO Operation
- FIFO Depth Calculations

# Save The **Palestinian** **Children**



**Rami Igra**  
Former Israeli  
Intelligence  
Official

**“Children in Gaza over 4  
deserve to be starved”**

[Former Mossad official: ‘Children in Gaza over 4  
deserve to be starved’ – Middle East Monitor](#)



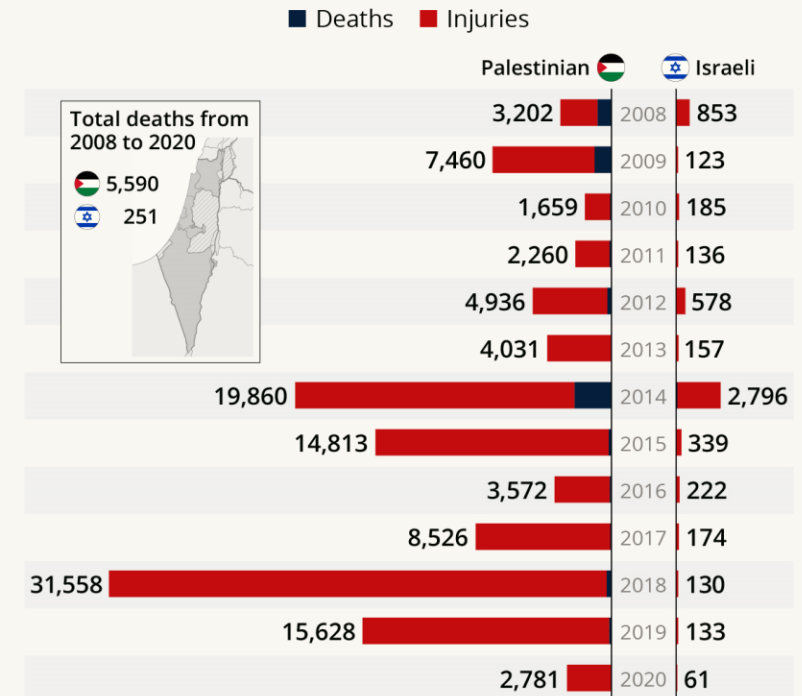
**Bezalel  
Smotrich**  
Israeli Minister  
of Finance

**“Might be ‘justified and  
moral’ to cause 2 million  
Gazans to die of hunger,  
but world won’t let us”**

[Israeli minister says it may be ‘moral’ to starve 2  
million Gazans, but ‘no one in the world would let  
us’ | CNN](#)

## The Human Cost Of The Israeli-Palestinian Conflict

Israeli & Palestinian deaths/injuries documented by the UN



Source: United Nations



statista

**Israel has been killing Palestinians long before Oct 7  
Do Palestinians have the right to defend themselves?**

# Clock Domain Crossing

Part 1

---

Amr Adel Mohammady

 /amradelm

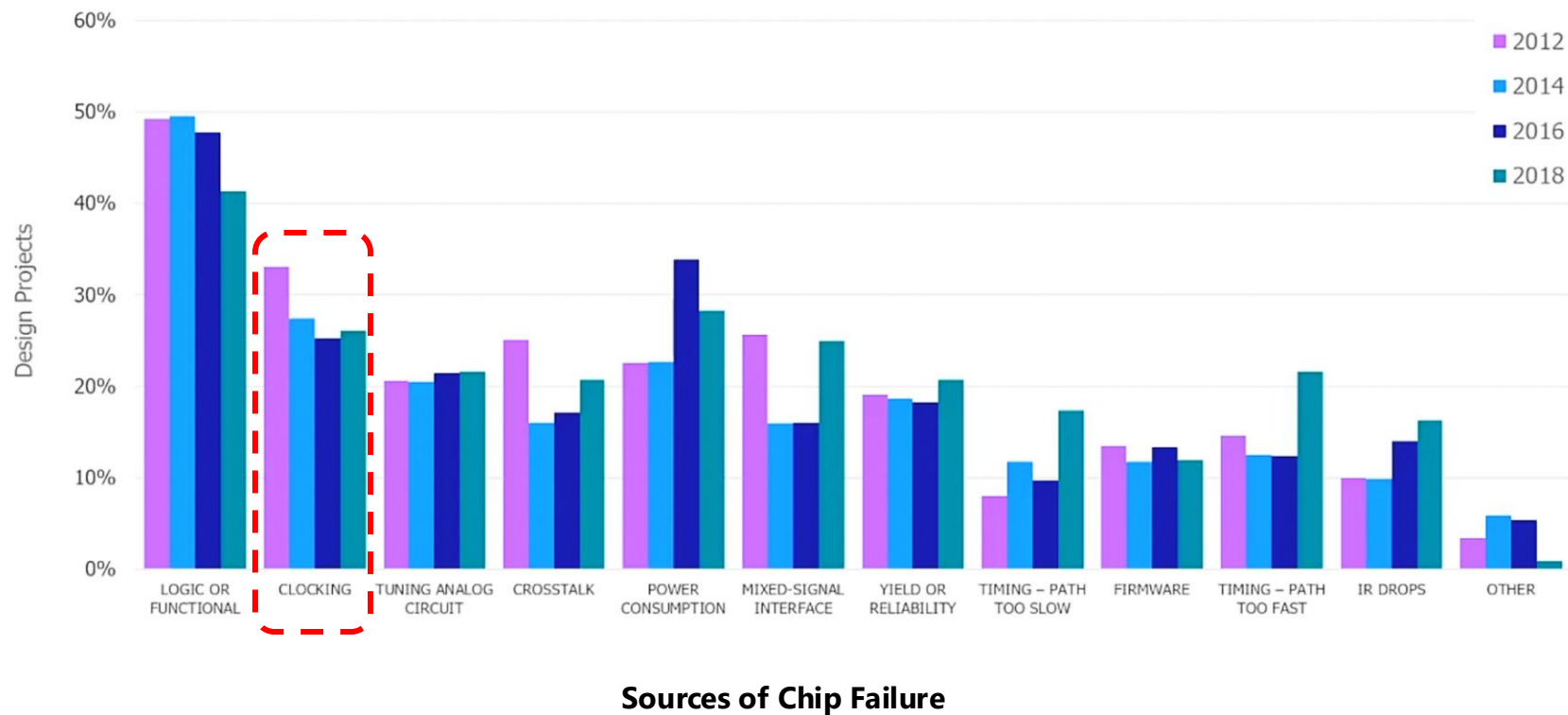
# Content

---

- Sources of chip failure
- How metastability happens in a flip-flop
- Synchronous vs asynchronous clock domains
- CDC concerns
- Mean Time Between Failure (MTBF)
  - Math derivation
  - Example
  - Adding multiple synchronizer stages
  - Summary of how to increase the MTBF

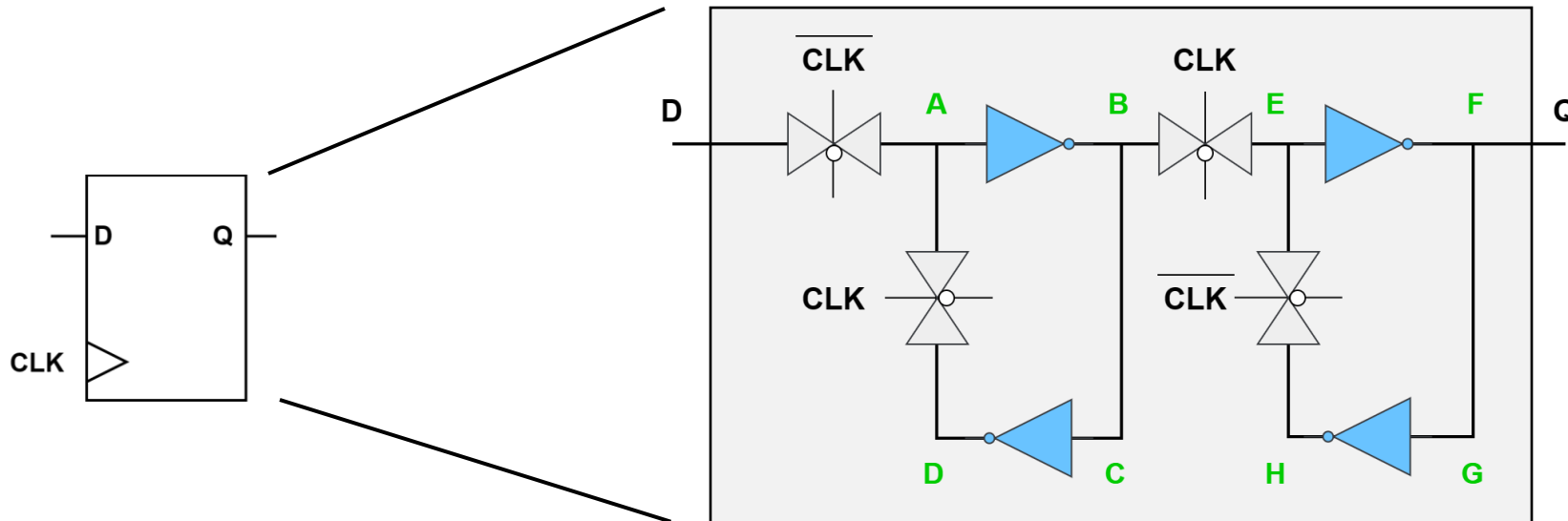
# Introduction

- Clock Domain Crossing (CDC) is a critical aspect of digital circuit design.
- In systems where multiple clock domains operate at different frequencies or phases when signals transfer from one clock domain to another, the risk of data corruption and metastability arises.
- CDC issues are a primary cause of chip failure. Improper handling of CDC can lead to timing errors, metastability, and data loss, impacting the chip's functionality and reliability.

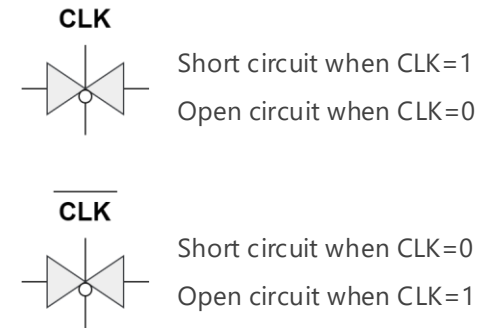


# Metastability in Sequential Circuits

- To understand metastability we need to look into the internal workings of a flip flop
- The diagram below shows one way to implement D flip flops using inverters and transmission gates.
- The transmission gates acts as a switch that opens or closes depending on a control signal
- The inverter loops are the storage elements that store the data

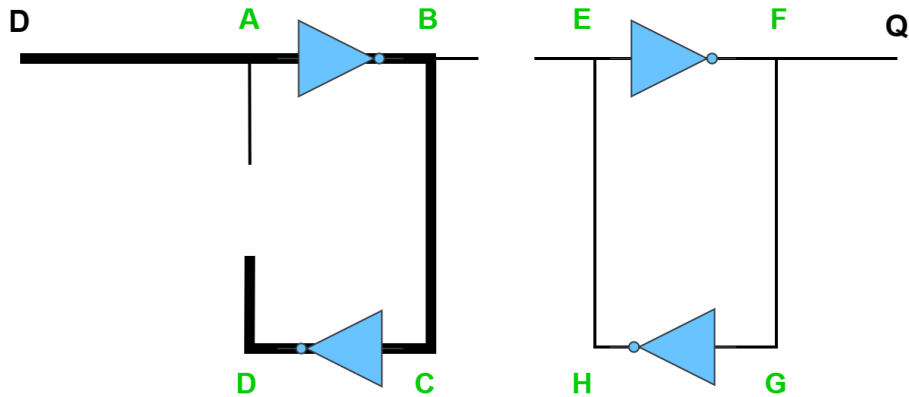


## Transmission Gates

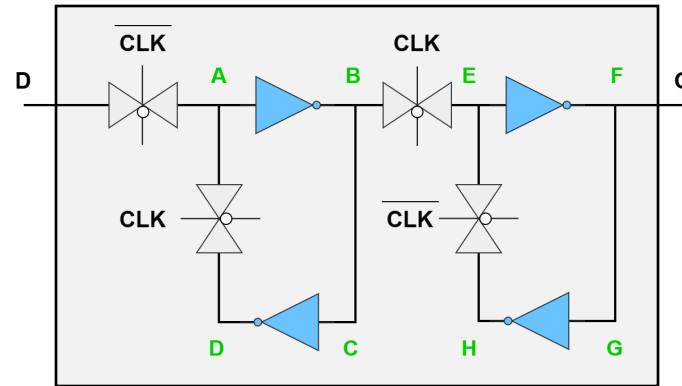
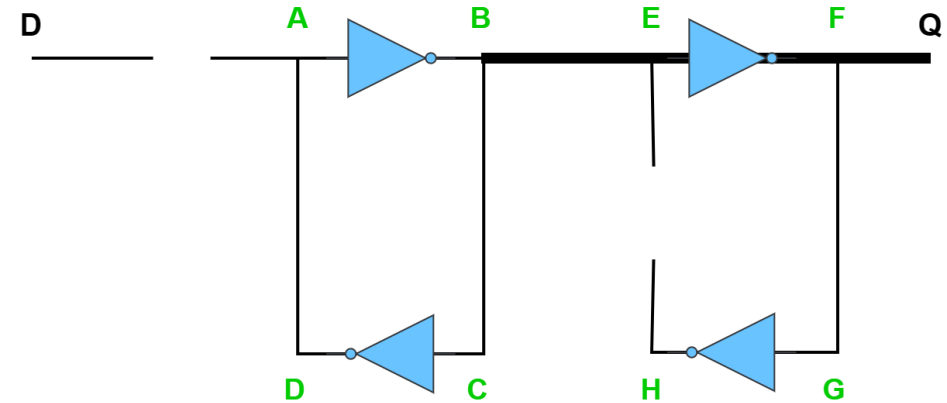


# Flip Flop Internal Operation

**1** Before the clock edge arrives (CLK=0), the input goes from the input pin D through **A-B-C-D** and waits for the clock edge.

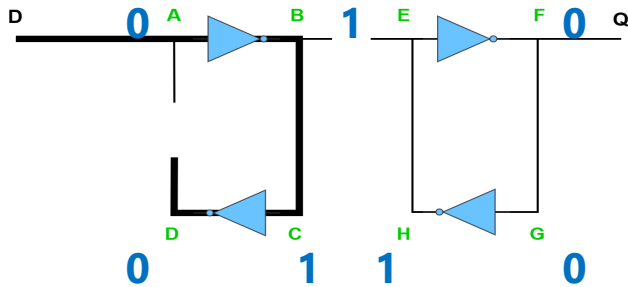


**2** After the clock edge arrives (CLK=1), The data flow through **B-E-F** to the output pin Q.

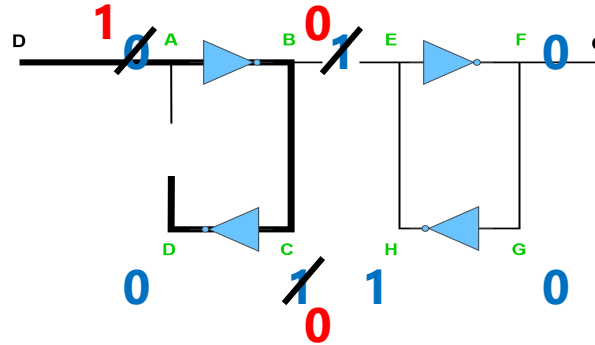


# Setup Time

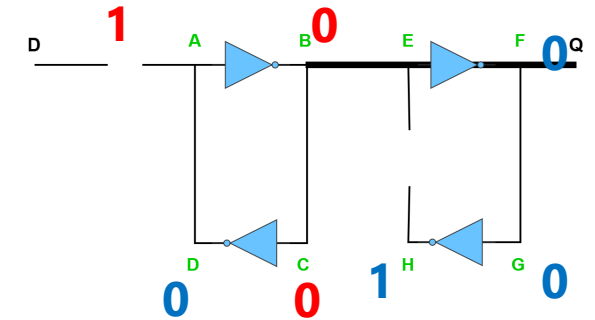
- 1** To understand how a setup violation happens lets go through this scenario:  
Lets assume the FF was storing a logic zero (0)



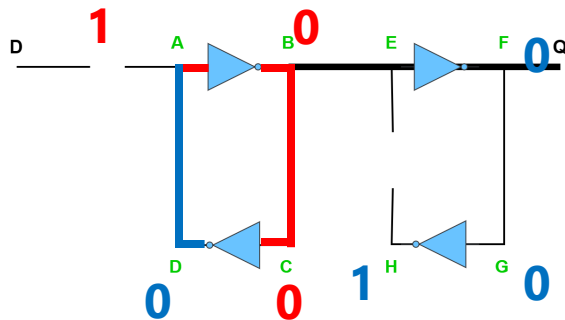
- 2** Now a **new data** arrives at the D input ,that is logic one (1), and starts overwriting the previous stored value



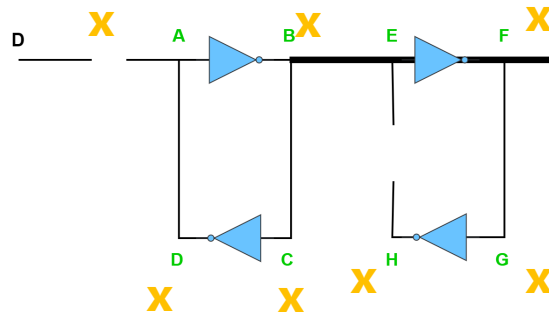
- 3** The clock edge arrives before the new data have time to overwrite node **D**. The transmission gates switch



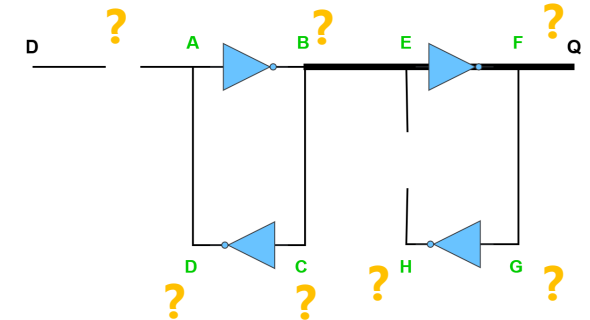
- 4** The transmission gate between **D** & **A** is now a short circuit, so **D** is trying to force the inverter loop to store the **old data** while **A-B-C** is trying to force the loop to store the **new data**



- 5** The conflict between the two electrical values will propagate to all the nodes in the FF and the output won't be a 0 or 1. The FF is said to be in a metastable state



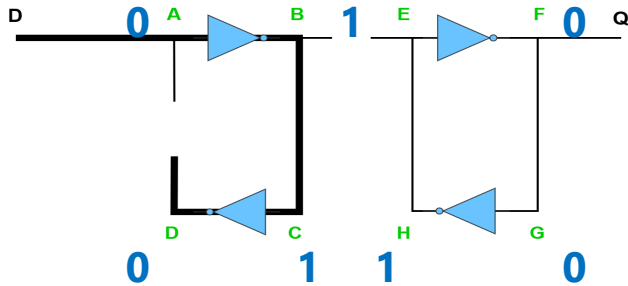
- 6** After some time one of the 2 values will overcome the other and the FF will leave the metastable state. The final state could be the **old data** or the **new data**



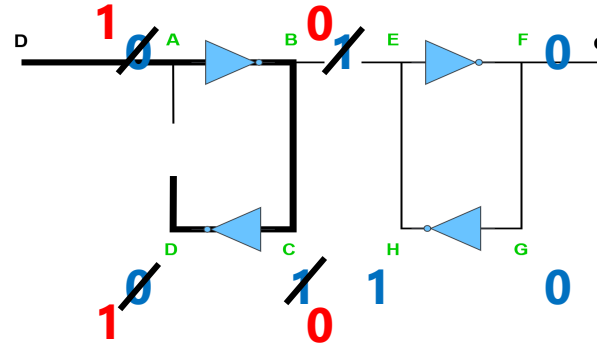


# Hold Time

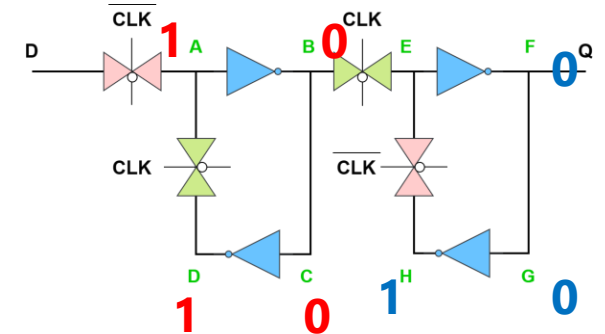
- 1** To understand how a hold violation happens lets go through this scenario:  
Lets assume the FF was storing a logic zero (0)



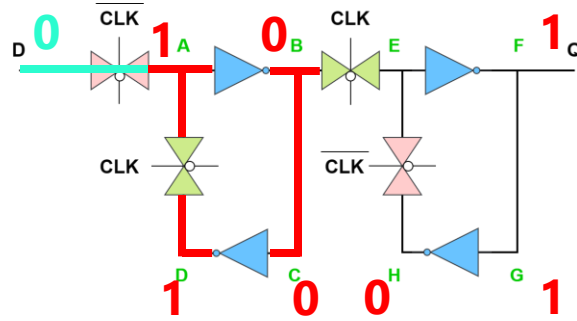
- 2** Now a **new data** arrives at the D input ,that is logic one (1), and starts overwriting the previous stored value



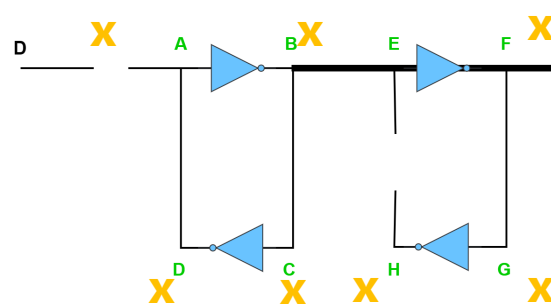
- 3** The clock edge arrives, the red transmission gates starts to open while the green ones starts to close (short)



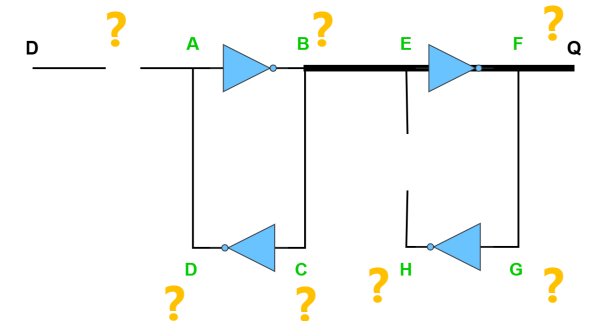
- 4** Before the red gates completely open, a **newer data** arrives at the D pin, The signal at the D pin is trying to force the inverter loops to store the **newer data**, the nodes **A-B-C-D** are trying to force it to store the **new data**



- 5** The conflict between the two electrical values will propagate to all the nodes in the FF and the output won't be a 0 or 1. The FF is said to be in a metastable state



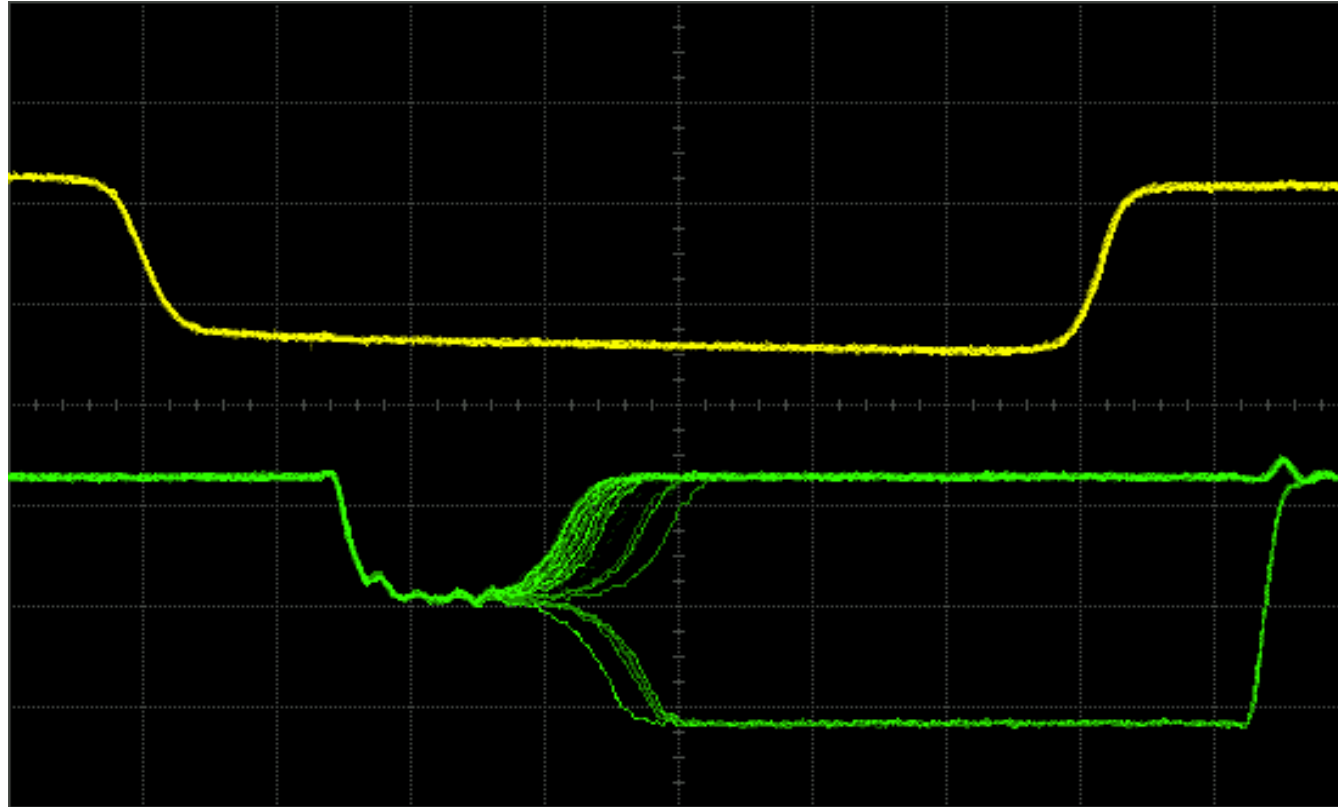
- 6** After some time one of the 2 values will overcome the other and the FF will leave the metastable state.



# Metastability

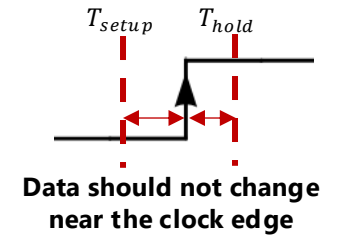
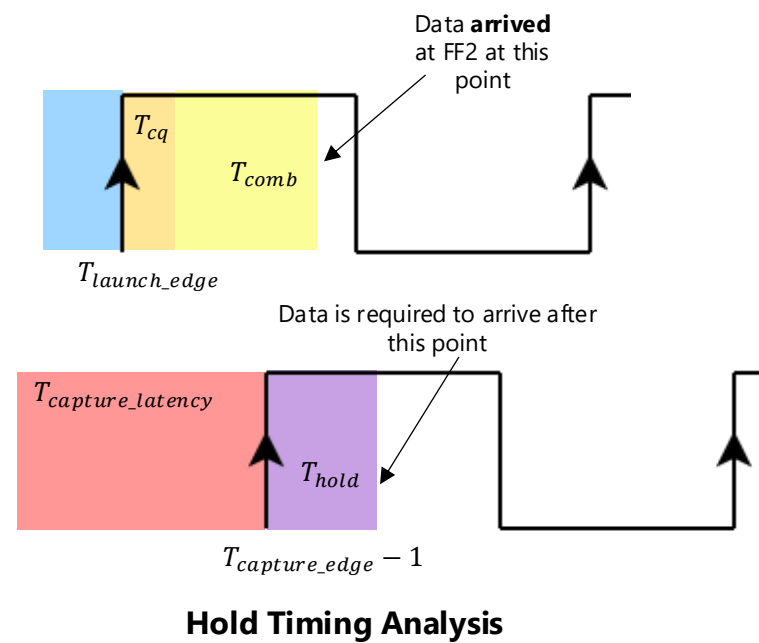
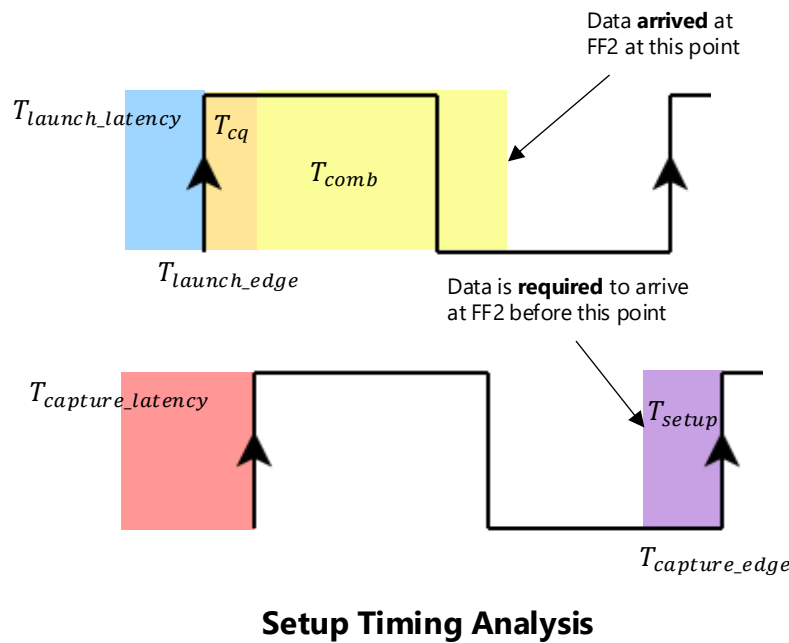
---

- The image below shows metastability in a FF. The FF becomes metastable (not 0 or 1) for some time before it settles to 0 or 1



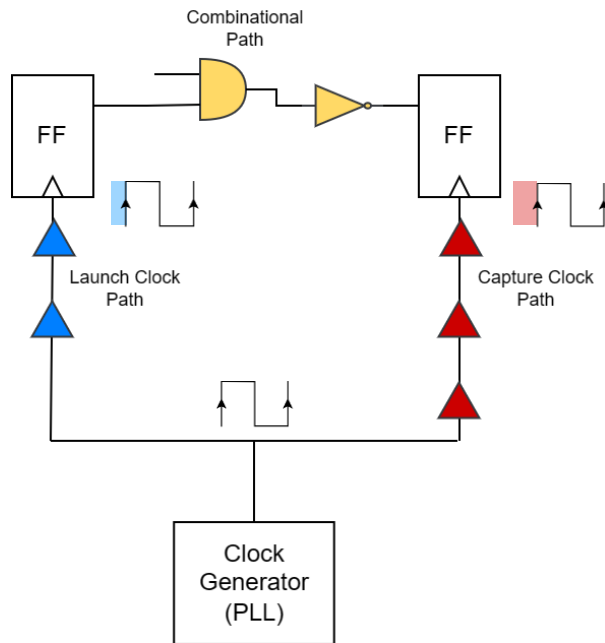
# Metastability in Sequential Circuits

- To avoid metastability we need to make sure the data doesn't change near the clock edge. We use static timing analysis (STA) to know when the data will arrive relative to the clock edge.
- If we can't know when the data will arrive relative to the clock edge, then we can't use STA.

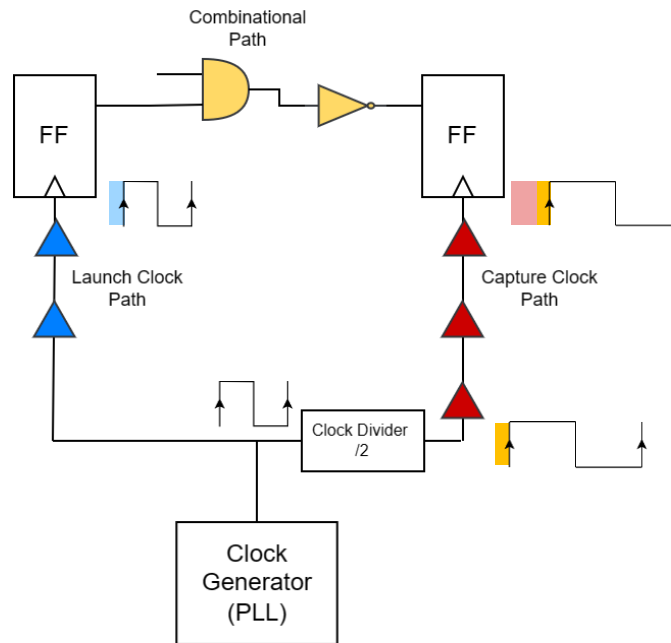


# Metastability in Sequential Circuits

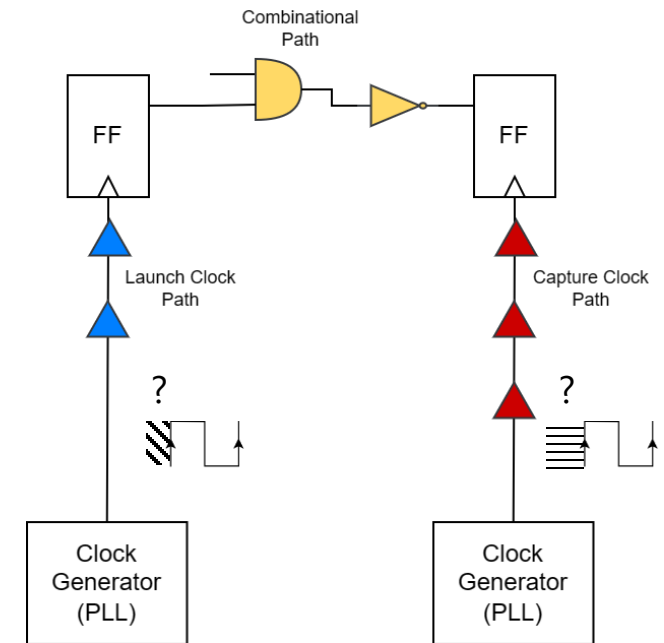
- When the clock of the launch and capture FFs come from the same source, we can know when the capture edge will arrive relative to the launch edge by calculating the difference in the clock network. This difference is called the skew or the phase difference.
- Even if the clock frequency is different between the launch and capture FFs, we still can use STA because we know the delay of the clock dividers and hence know the phase difference.
- The problem arises when the clocks of the launch and capture FFs come from different sources (PLL, oscillator). There is no way to know the phase difference between the two PLLs even if they have the same frequency. We can't use STA and we need to find another way to deal with this issue.



**Known Phase Difference**



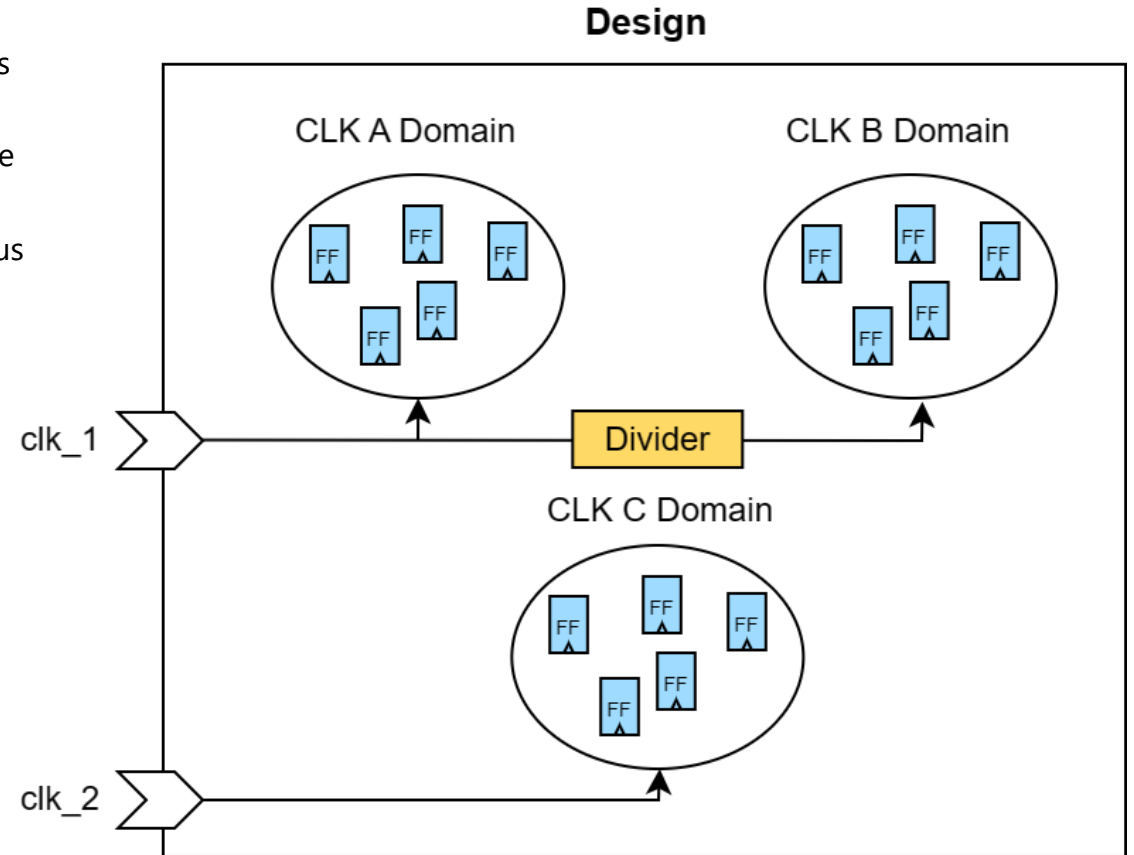
**Known Phase Difference**



**Unknown Phase Difference**

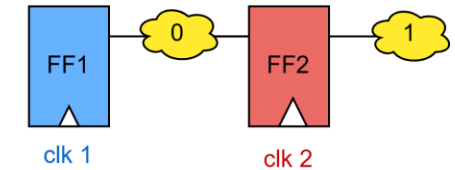
# Clock Domains

- **Clock domain** is a group of registers that uses the same clock.
- **Synchronous Clocks:** Are clocks where the phase difference between the clocks is known ,such as a clock and the divided version of it.
- **Asynchronous Clocks:** Are clocks that come from different sources and the phase difference is not known.
- Clock domain crossing (CDC) cares about signals that travel between asynchronous clock domains

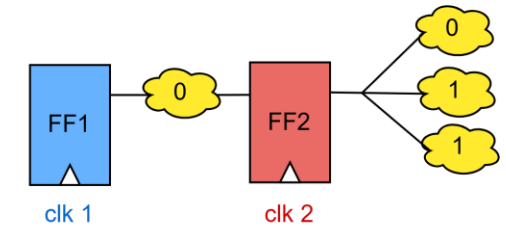


# CDC Concerns

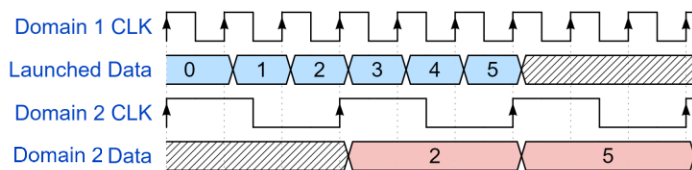
- Before we discuss the solutions to CDC we need to know the issues we need to solve.
- **CDC Concerns:**
  - Data corruption: When metastability occurs the data might get corrupted since it can settle at 0 or 1 regardless of the input.
  - Data incoherence: The system might be fault tolerant and handle corrupted data. However, we want all the blocks in the circuit to see the same data (either all see 0 or all see 1)
  - Data loss: When data goes from a faster domain to a slower domain some samples might be skipped/lost
  - Data duplication: When a signal is intended to occur for one cycle but is read multiple times by the receiving domain.
  - Chip burning: During metastability, both the PMOS and NMOS networks are ON. This results in a big current flowing through the transistor. If multiple gates entered this state at the same time and for a long duration the chip might burn.



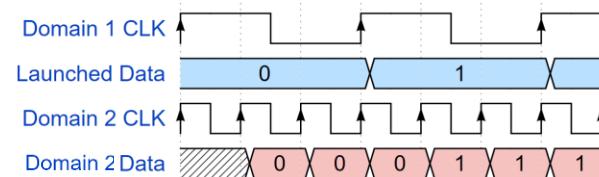
**Data Corruption**



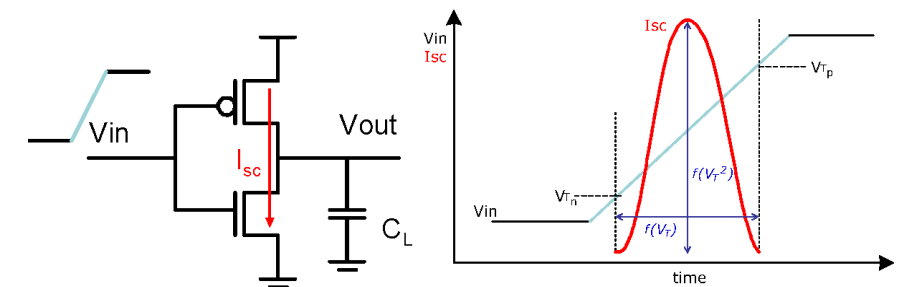
**Data Incoherence**



**Data Loss**



**Data Duplication**



**Short Circuit Current<sup>1</sup>**

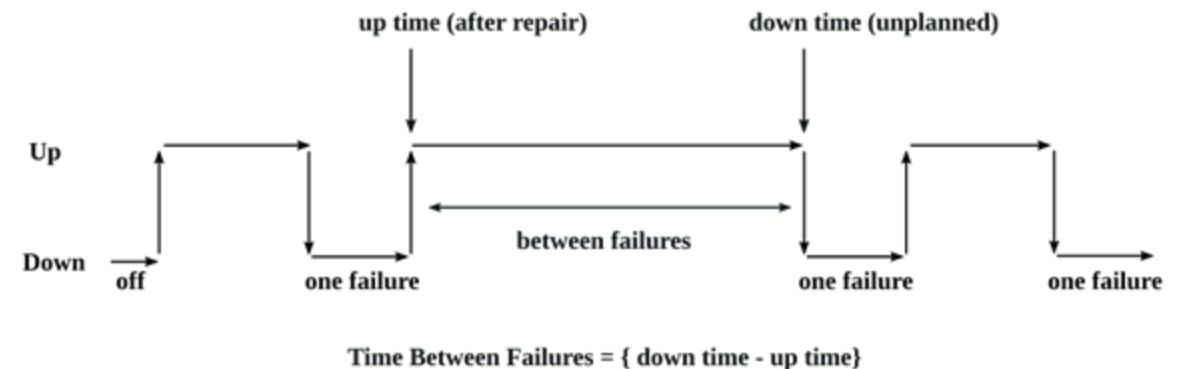
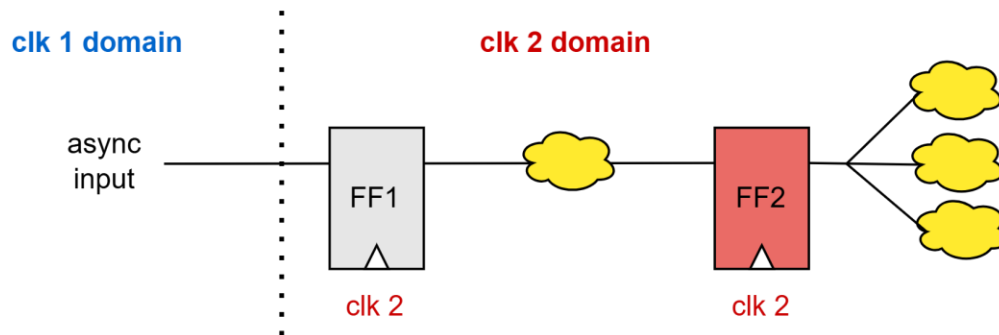
---

# Mean Time Between Failure (MTBF)

# Mean Time Between Failure

**"We must embrace pain and burn it as fuel for our journey." — Kenji Miyazawa**

- Our first step in dealing with CDC metastability is to realize that we can't avoid it.
- Instead, we will try to handle and mitigate the problems that arise due to it.
- We will accept that FF1 (in the left diagram) will go metastable but we will do our best to make sure the metastability value doesn't reach **FF2**. Otherwise, the metastable value will reach multiple areas in the chip causing it to enter a faulty state or worse burn the chip.
- Mean Time Between Failure (MTBF) is a reliability metric used to predict the average time between failures for a system or component during normal operation.
- So, our 2<sup>nd</sup> step in dealing with CDC is to do our best to decrease the frequency of error or in other words: increase the MTBF.
- In the following slides we will derive the mathematical expression for MTBF and then see how to increase it.





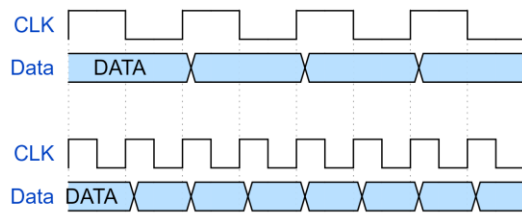
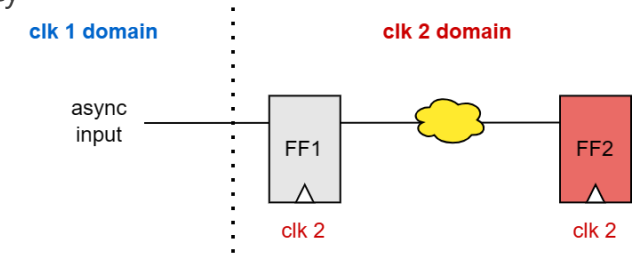
# MTBF Derivation

- We need to know the frequency that **FF2** will receive a metastable value and then do our best to decrease this frequency

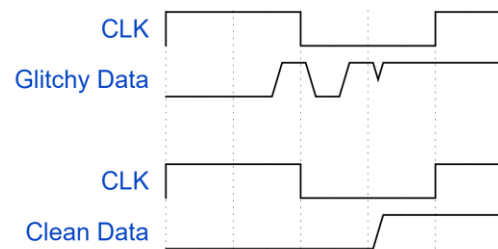
- To calculate this we need to know 2 things:

1. How frequently will **FF1** go metastable?  $f_{meta}$

- FF1** will go metastable if a toggle from clock domain 1 reaches it during the setup and hold window.
- This depends on several factors:
  - Clock 1 frequency  $f_{clk1}$** : The higher the frequency the more we receive inputs and therefore toggles
  - Activity factor  $\alpha$** : This term shows how frequent does the input toggle relative to the clock.  $\alpha = 1$  means the input toggle once every clock cycle.  $\alpha > 1$  means the input toggles multiple times every clock cycle (glitches).
  - The setup and hold window  $T_0$** <sup>[1]</sup>: If clk 2 period is  $T_{clk2}$ , the probability that the async input will arrive during the setup and hold window =  $\frac{T_0}{T_{clk2}}$
- Therefore  $f_{meta} = \text{frequency of toggles } (f_d) \times \text{probability that the toggle will cause metastability} = \alpha f_{clk1} \times \frac{T_0}{T_{clk2}} = \alpha f_{clk1} f_{clk2} T_0$



Higher Freq -> More Toggles

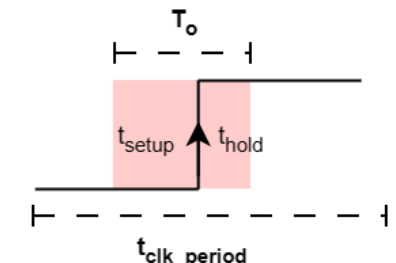


More Glitches -> More Toggles

$$\alpha = 3$$

$$\alpha = 1$$

$$f_d(\text{Toggle Frequency}) = \alpha f_{clk1}$$



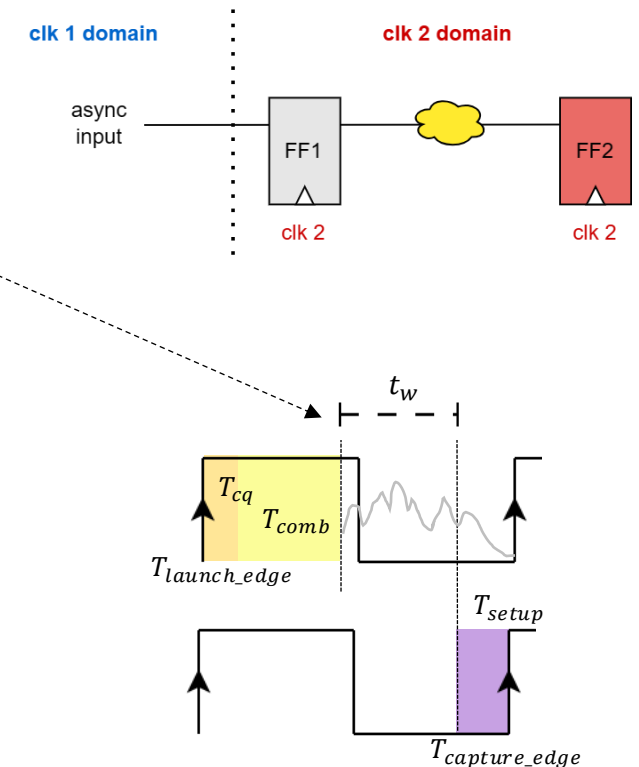
[1]

[1]: There are few resources explaining this factor in detail. Reference (5) mentions it's the FF propagation delay  $T_{cq}$ . While reference (3, 4) says it's the metastability window

# MTBF Derivation

2. The 2<sup>nd</sup> thing to know is: If **FF1** went metastable, what is the probability that the data will remain metastable till reaching **FF2**?  $P_u$

- This is given by  $P_u = e^{-\frac{t_w}{\tau_{tech}}}$ , where
  - **Time window  $T_w$** : The time allowed for the metastable value to settle. This is the positive slack of the path between FF1 and FF2.
  - **Time constant  $\tau_{tech}$** : A time constant that depends on how fast the logic circuit drives the output towards a valid logic level. This is a function of RC time constants and circuit gain.



• Therefore, the frequency that **FF1** will produce metastable value for **FF2** =  $f_{meta} \times P_u = \alpha f_{clk1} f_{clk2} T_o \times e^{-\frac{t_w}{\tau_{tech}}}$

• The time between two failures (MTBF) =  $\frac{1}{\text{frequency of failure}} = \frac{1}{\alpha f_{clk1} f_{clk2} T_o \times e^{-\frac{t_w}{\tau_{tech}}}} = \frac{e^{\frac{t_w}{\tau_{tech}}}}{\alpha f_{clk1} f_{clk2} T_o}$

• **To increase the MTBF:**

- Use lower frequencies  $f_{clk1} f_{clk2}$
- Decrease the activity factor  $\alpha$ :
- Decrease  $T_o$ :
- Increase  $T_w$

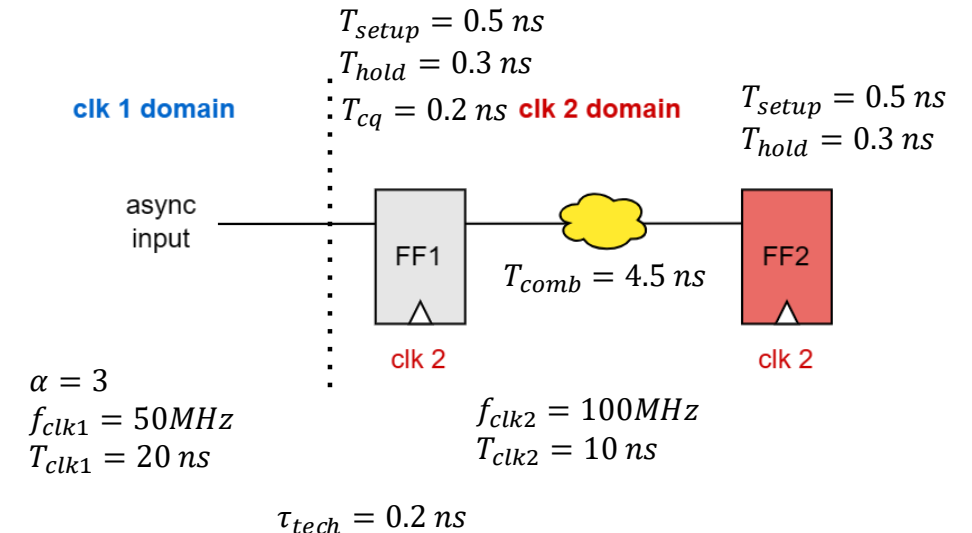
# MTBF Calculations – Example

- To get a feeling of the impact of each term in the equation we will work out an example using the values in the diagram below:

- $t_w = \text{slack between FF1, FF2} = \text{clk2 period} - T_{cq} - T_{comb} - T_{setup} = 10 - 0.2 - 4.5 - 0.5 = 4.8 \text{ ns}$
- $T_o = \text{setup time} + \text{hold time} = 0.5 + 0.3 = 0.8 \text{ ns}^1$
- $$MTBF = \frac{1}{\alpha f_{clk1} f_{clk2} T_o} e^{\frac{t_w}{\tau_{tech}}} = \frac{1}{3 \times (50 \times 10^6) (100 \times 10^6) (0.8 \times 10^{-9})} e^{\frac{4.8 \times 10^{-9}}{0.2 \times 10^{-9}}} = 2.207 \times 10^3 \text{ seconds} = 0.0255 \text{ days}$$

- Lets try increasing the MTBF by using fast FFs with better  $T_{cq}$ ,  $T_{setup}$ ,  $T_{hold}$ . This will enhance  $t_w$  and  $T_o$  and both will enhance MTBF

- The new FF has :  $T_{cq} = 0.05$ ,  $T_{setup} = 0.2$ ,  $T_{hold} = 0.1$
- New  $t_w = 10 - 0.05 - 4.5 - 0.2 = 5.25 \text{ ns}$
- New  $T_o = 0.2 + 0.1 = 0.3 \text{ ns}$
- $$MTBF = \frac{1}{3 \times (50 \times 10^6) (100 \times 10^6) (0.3 \times 10^{-9})} e^{\frac{5.25 \times 10^{-9}}{0.2 \times 10^{-9}}} = 5.585 \times 10^4 \text{ seconds} = 0.646 \text{ days}$$
- We got x25 improvement by using fast FFs but it's still not enough.



[1]: There are few resources explaining this factor in detail. Reference (5) mentions it's the FF propagation delay  $T_{cq}$ . While reference (3, 4) says it's the metastability window

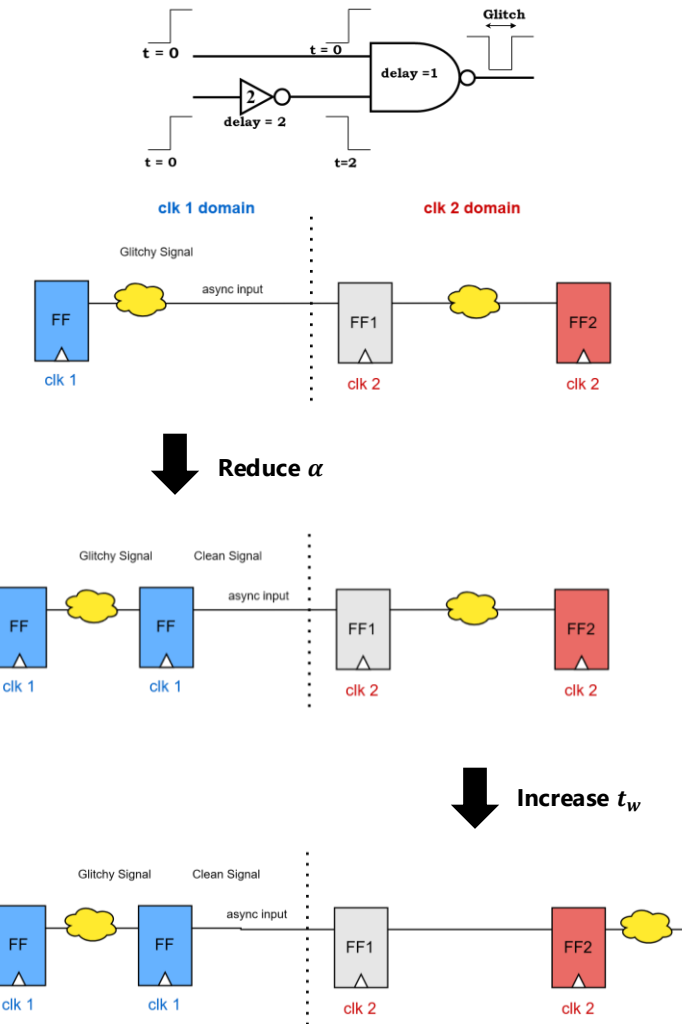
# MTBF Calculations – Example

- **Lets focus on the activity factor  $\alpha$ . To reduce it we need to reduce the glitches.**

- Glitches happen due to unequal delays in combinational circuits.
- The easiest way to get a clean signal without glitches is to add a register. The register changes value only at the clock edge and so, acts as a filter against glitches. This will lead to an activity factor  $\alpha \leq 1$
- We will use the worst case  $\alpha = 1$  where the register toggles every clock cycle.
- $$MTBF = \frac{1}{1 \times (200 \times 10^6)(300 \times 10^6)(0.3 \times 10^{-9})} e^{\frac{5.25 \times 10^{-9}}{0.2 \times 10^{-9}}} = 1.675 \times 10^5 \text{ seconds} = 1.94 \text{ days}$$

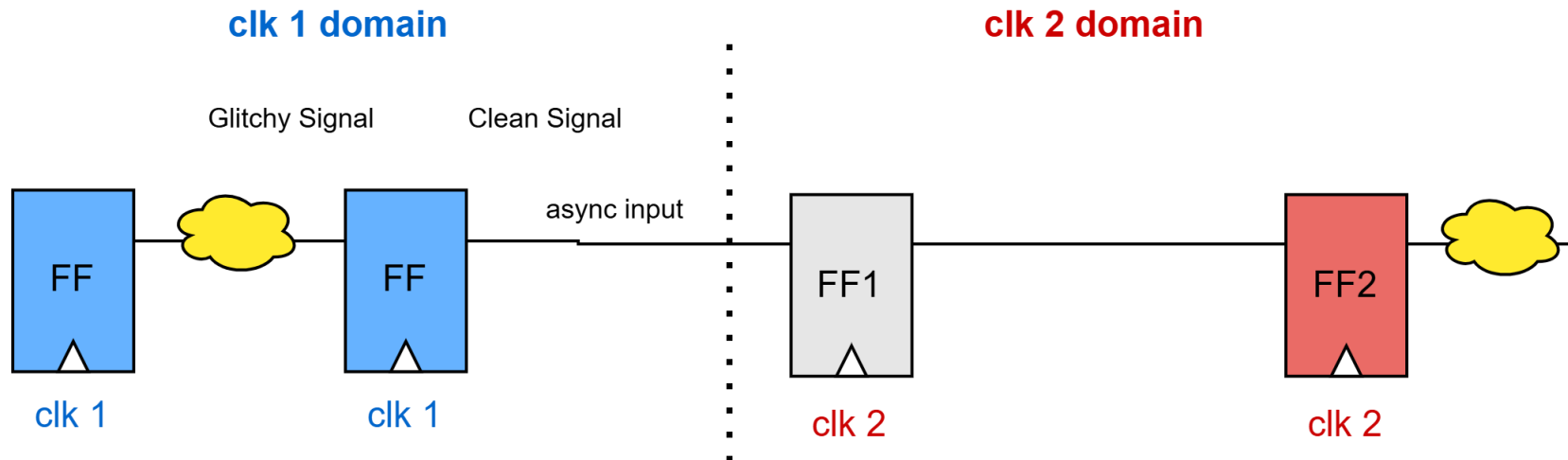
- **Now lets enhance the slack  $t_w$ . We enhanced it when we used faster FFs. Now we will focus on the combinational path delay.**

- We can use fast cells or reduce the wire delay between the cells.
- However, the optimal solution is to remove the delay entirely by moving the logic after FF2 making  $T_{comb} = 0$
- Therefore,  $t_w = 10 - 0.05 - 0.2 = 9.75 \text{ ns}$
- $$MTBF = \frac{1}{1 \times (200 \times 10^6)(300 \times 10^6)(0.3 \times 10^{-9})} e^{\frac{9.75 \times 10^{-9}}{0.2 \times 10^{-9}}} = 9.903 \times 10^{14} \text{ seconds} = 1.1454 \times 10^9 \text{ days} = 3.137 \times 10^6 \text{ years}$$
- Removing the logic leads to a massive improvement



# MTBF Calculations – Example

- **Lets assume the architecture team decided to increase the frequency of  $f_{clk2} = 300MHz$  ( $T_{clk2} = 3.3\text{ ns}$ ). This will also affect  $T_w$** 
  - $t_w = 3.3 - 0.05 - 0.2 = 3.05\text{ ns}$
  - $MTBF = \frac{1}{1 \times (50 \times 10^6) (300 \times 10^6) (0.3 \times 10^{-9})} e^{\frac{3.05 \times 10^{-9}}{0.2 \times 10^{-9}}} = 0.9\text{ seconds}$
  - The MTBF decreased significantly

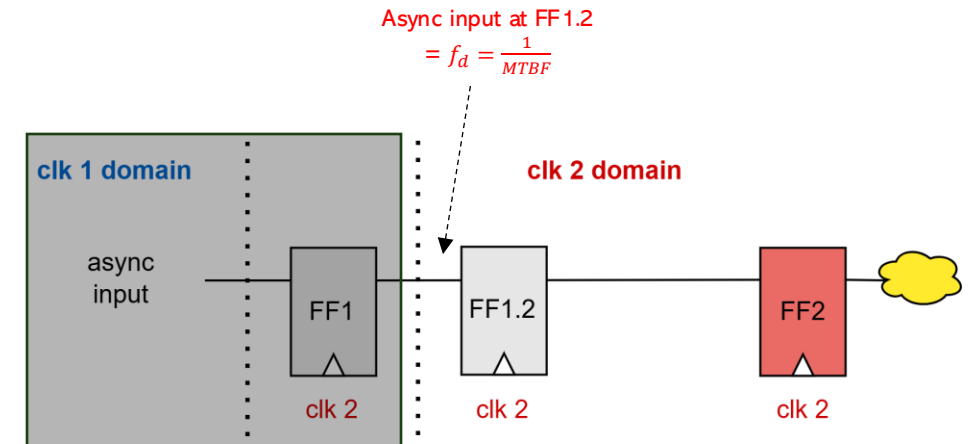


# MTBF Calculations – Two Flip Flops

- Now let's add a 2<sup>nd</sup> FF after **FF1** and see how this will affect the MTBF for **FF2**.
  - We will assume clock domain 1 along with **FF1** are sources of a toggling input and see how this input affects **FF1.2** and therefore **FF2**
  - We have all the values to substitute in the MTBF equation except the frequency of an asynchronous input ( $f_d = \alpha f_{clk2}$ ) arriving at **FF1.2**
  - We showed that the frequency that **FF1** produces a metastable value =  $f_{meta} \times P_u = \alpha f_{clk1} f_{clk2} T_o \times e^{-\frac{t_w}{\tau_{tech}}}$ . We can assume this is the frequency that **FF1** produces a toggle  $f_{d1}$ .
  - Therefore the frequency that **FF2** receives a metastable output =  $f_{meta} \times P_u = f_{d1} f_{clk2} T_o \times e^{-\frac{t_w}{\tau_{tech}}} = \alpha f_{clk1} f_{clk2}^2 T_o^2 \times e^{-\frac{t_{w1}}{\tau_{tech}} - \frac{t_{w2}}{\tau_{tech}}}$ .
  - $$MTBF = \frac{e^{\frac{t_{w1}}{\tau_{tech}} + \frac{t_{w2}}{\tau_{tech}}}}{\alpha f_{clk1} f_{clk2}^2 T_o^2}$$
  - Now if we add a 2nd FF:**
    - $$MTBF_2 = \frac{1}{f_d f_{clk2} T_o} e^{\frac{t_w}{\tau_{tech}}} = \frac{1}{\frac{1}{MTBF_1} f_{clk2} T_o} e^{\frac{t_w}{\tau_{tech}}} = \frac{1}{0.9(300 \times 10^6)(0.3 \times 10^{-9})} e^{\frac{3.05 \times 10^{-9}}{0.2 \times 10^{-9}}}$$

$$= 4.2 \times 10^7 \text{ seconds} = 485.69 \text{ days}$$
    - The improvement is good but not enough
  - If we add a third FF:**
    - $$MTBF_3 = \frac{1}{f_d f_{clk2} T_o} e^{\frac{t_w}{\tau_{tech}}} = \frac{1}{\frac{1}{MTBF_2} f_{clk2} T_o} e^{\frac{t_w}{\tau_{tech}}} = \frac{1}{\frac{1}{4.2 \times 10^7} (300 \times 10^6)(0.3 \times 10^{-9})} e^{\frac{3.05 \times 10^{-9}}{0.2 \times 10^{-9}}}$$

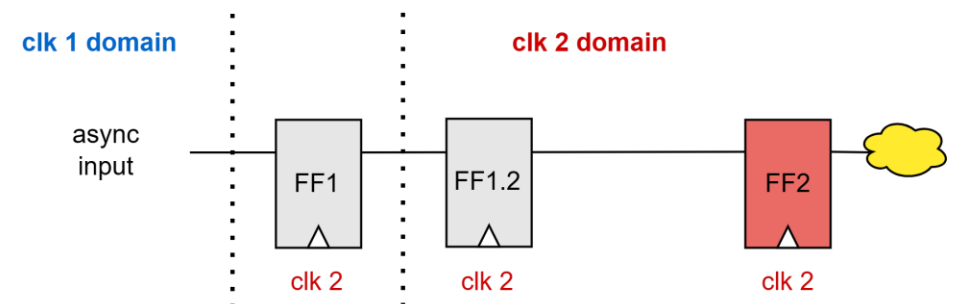
$$= 1.96 \times 10^{15} \text{ seconds} = 2.2685 \times 10^{10} \text{ days} = 6.211 \times 10^7 \text{ years}$$



# How to Increase the MTBF

- **From the previous calculations we can have several observations:**

- Adding more stages/FFs enhances the MTBF.
  - For low frequency designs 2 stages are enough. But for high-frequency designs, we need to add more than 2 stages.
  - The disadvantage of this solution is the added latency due to the added stages.
  - The FFs stages (FF1, and FF1.2) are called **CDC synchronizers**
- Reducing  $T_{comb}$  gets better MTBF. To do this:
  - Don't add any combinational logic in front of the synchronizers
  - In the PNR stage, place the sync FF close to each other to minimize the buffering and wire delay.
- Using fast FFs with smaller  $T_o$ : Standard cell libraries contain special sync flip flops that have these parameters enhanced to get better MTBF.
- Reducing the activity factor by sampling the data from the sending domain after a FF to avoid glitches



# Conclusion

---

- **We want to know what we solved till now. Our CDC concerns were:**
  - Data corruption: **Not fixed**. The system, till now, has to be fault tolerant and be able to handle corrupted data.
  - Data incoherence: **Fixed**. The metastable value settles within the synchronizers at 0 or 1 and then propagates to all the 2<sup>nd</sup> domain blocks with the same settled value
  - Data loss: **Not fixed**
  - Data duplication: **Not fixed**
  - Chip burning: **Fixed**. We limited the metastability propagation between the synchronizers and reduced its occurrence frequency.
- In the next parts we will see how to deal with the remaining concerns.



# References

---


- 1) <https://www.uio.no/studier/emner/matnat/ifi/IN3160/v21/timeplan/in3160-l92-clock-domains.pdf>
- 2) <https://ieeexplore.ieee.org/document/1676187>
- 3) <https://www.edn.com/keep-metastability-from-killing-your-digital-design/>
- 4) <https://people.ece.ubc.ca/~edc/7660.jan2018/lec11.pdf>
- 5) <https://www.onsemi.com/pub/Collateral/AN1504-D.PDF>

# Clock Domain Crossing

Part 2

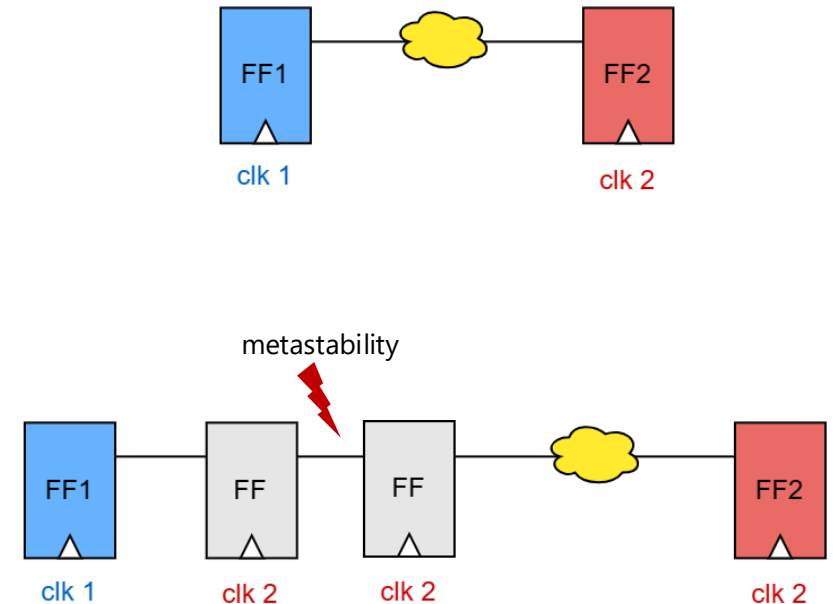
---

Amr Adel Mohammady

 /amradelm

# Introduction

- In the previous part we saw how to limit the effect of metastability between multiple stages of flip-flop that we called CDC synchronizer
- The idea is to isolate the metastable value between the synchronizer FFs and give it enough time until it settles to a known value then give it to the receiving domain
- We also discussed the MTBF metric and how to increase it to ensure a reliable circuit.
- **We want to know what we solved till now. Our CDC concerns were:**
  - Data corruption: **Not fixed**. The system, till now, has to be fault tolerant and be able to handle corrupted data.
  - Data incoherence: **Fixed**. The metastable value settles within the synchronizers at 0 or 1 and then propagates to all the blocks of domain 2 with the same settled value
  - Data loss: **Not fixed**
  - Data duplication: **Not fixed**
  - Chip burning: **Fixed**. We limited the metastability propagation between the synchronizers and reduced its occurrence frequency.
- In this part we will handle some of the remaining concerns

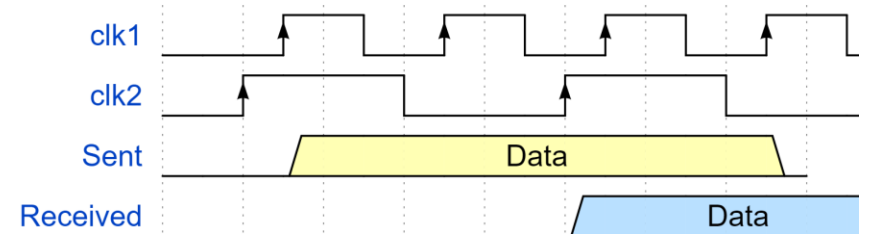
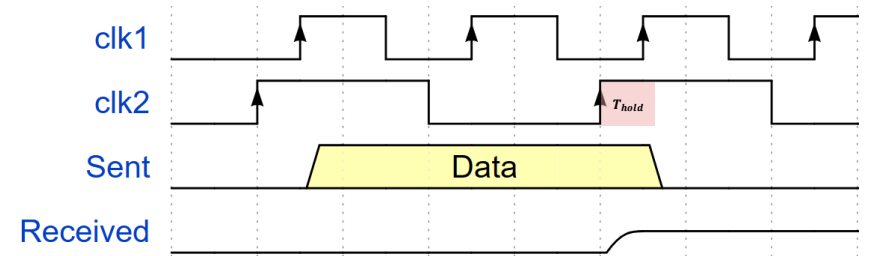
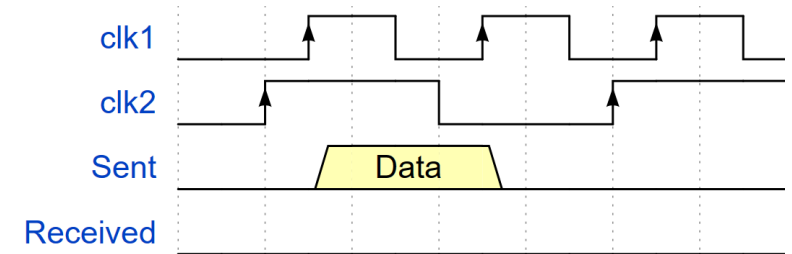


---

# Required Pulse Width

# Required Pulse Width

- Adding synchronizers only protected the chip from seeing a metastable value. However, we still get corrupted data because the data can settle at a value different than the intended one.
- To ensure that correct data is received we need to make sure the data pulse is wide enough to be safely captured by a domain 2 clock edge.
- The first example shows a small data width (one clock wide) that completely got missed by any capture edge of clock 2
- The second example shows a wider data pulse that reached a capture edge but not wide enough that it caused a hold violation and therefore metastability
- The required width for correct operation is  $\geq \text{clk2 period} + T_{\text{setup}} + T_{\text{hold}}$ . This way we guarantee a capture edge will lie within the data pulse width without any setup or hold violation.
- As a rule of thumb<sup>1</sup>, we need to make the data at least  $1.5 \times \text{clk2 period}$  wide.



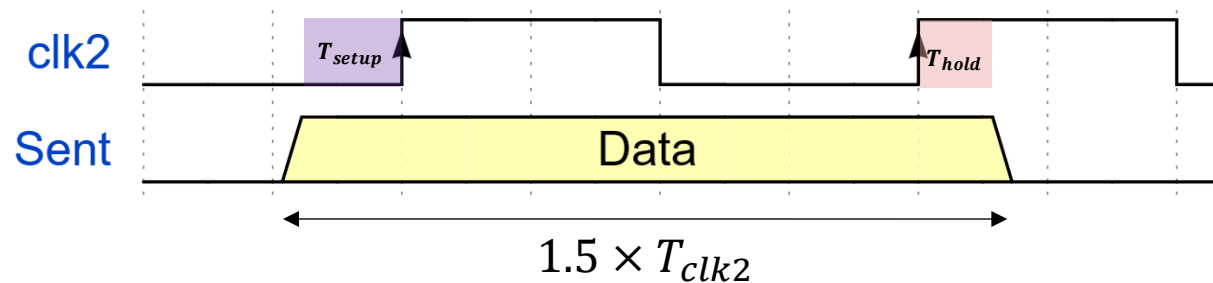
# Required Pulse Width – Examples

**1. (FAST TO SLOW) Let  $f_{clk1} = 225 \text{ MHz}$ ,  $f_{clk2} = 150 \text{ MHz}$ . For how many cycle should domain 1 hold the data stable to guarantee safe capture by domain 2?**

- $T_{clk1} = \frac{1}{225 \times 10^6} = 4.4 \text{ ns}$ .  $T_{clk2} = \frac{1}{150 \times 10^6} = 6.6 \text{ ns}$ .
- The data should be held stable for  $1.5 \times T_{clk2} = 1.5 \times 6.6 = 9.9 \text{ ns}$
- The number of cycles =  $\frac{9.9}{4.4} = 2.25 \text{ cycles} \cong 3 \text{ cycles}$ .

**2. (SLOW TO FAST) Let  $f_{clk1} = 100 \text{ MHz}$ ,  $f_{clk2} = 120 \text{ MHz}$ . For how many cycle should domain 1 hold the data stable to guarantee safe capture by domain 2?**

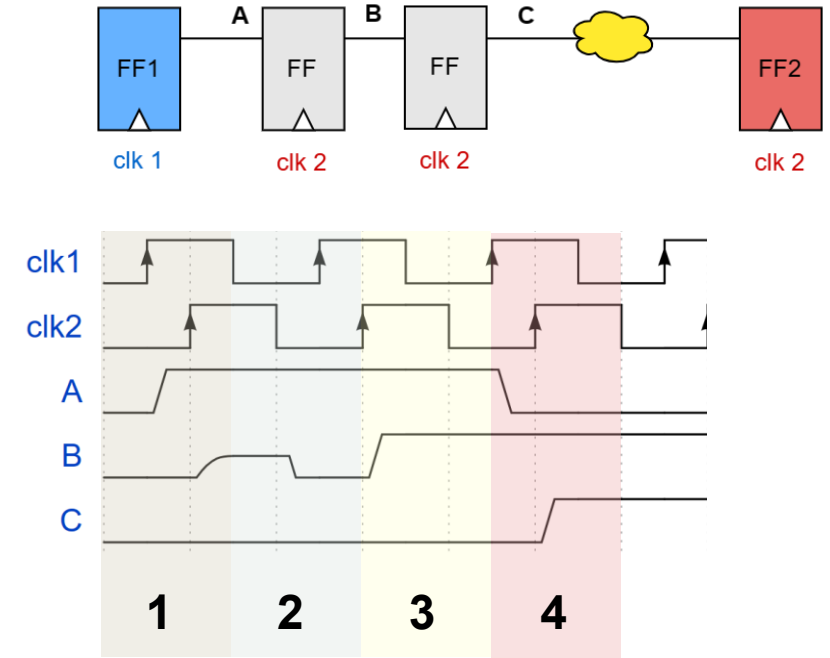
- $T_{clk1} = \frac{1}{100 \times 10^6} = 10 \text{ ns}$ .  $T_{clk2} = \frac{1}{120 \times 10^6} = 8.3 \text{ ns}$ .
- The data should be held stable for  $1.5 \times T_{clk2} = 1.5 \times 8.3 = 12.45 \text{ ns}$
- The number of cycles =  $\frac{12.45}{10} = 1.245 \text{ cycles} \cong 2 \text{ cycles}$ .



# Required Pulse Width

- Consider the example on the right:

- Domain 1 sends signal **A** to domain 2. The change occurs close to the edge of clk2 causing a metastability in the first sync FF.
  - Signal **B** leaves metastability and settle at a different value "logic 0".
  - C** gets the wrong value "logic 0". However, because pulse **A** was wide enough, it met another capture edge of clk2 without violating setup or hold. The correct value enters the first sync FF.
  - The logic receives the correct value "logic 1".
- This shows we can safely transfer a pulse from one domain to another provided that the pulse is wide enough.



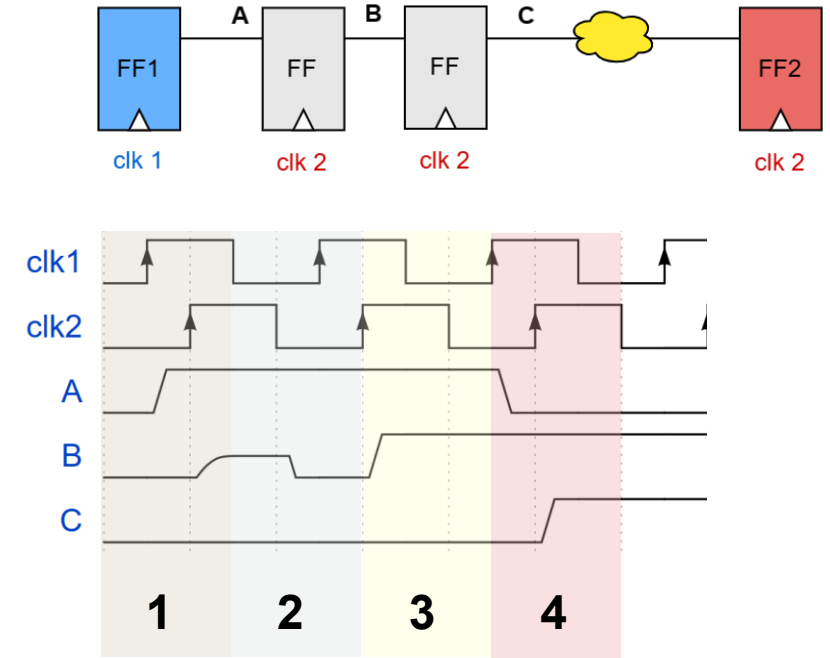
# The Issue of Varying Delays/Settling Time

- Consider the same previous example :

1. Domain 1 sends signal **A** to domain 2. The change occurs close to the edge of clk2 causing a metastability in the first sync FF.
2. Signals **B** leaves metastability and settle at a different value "logic 0".
3. **C** gets the wrong value "logic 0". However, because pulse **A** was wide enough, it met another capture edge of clk2 without violating setup or hold. The correct value enters the first sync FF.
4. The logic receives the correct value "logic 1".

- Lets consider another case were **B** in cycle (2) have metastability but settles at "logic 1" which happens to be the correct logic:

- This means the pulse will arrive at **C** earlier one cycle, that is at cycle (3) instead of (4).
- The correct logic have a varying delay: it can arrive at (3) or (4).
- This varying delay is unavoidable and the design needs to be tolerant of such delays and work under either cases
- Advanced CDC tools inject varying delays into the CDC path and verify the design is functionally correct in all cases.



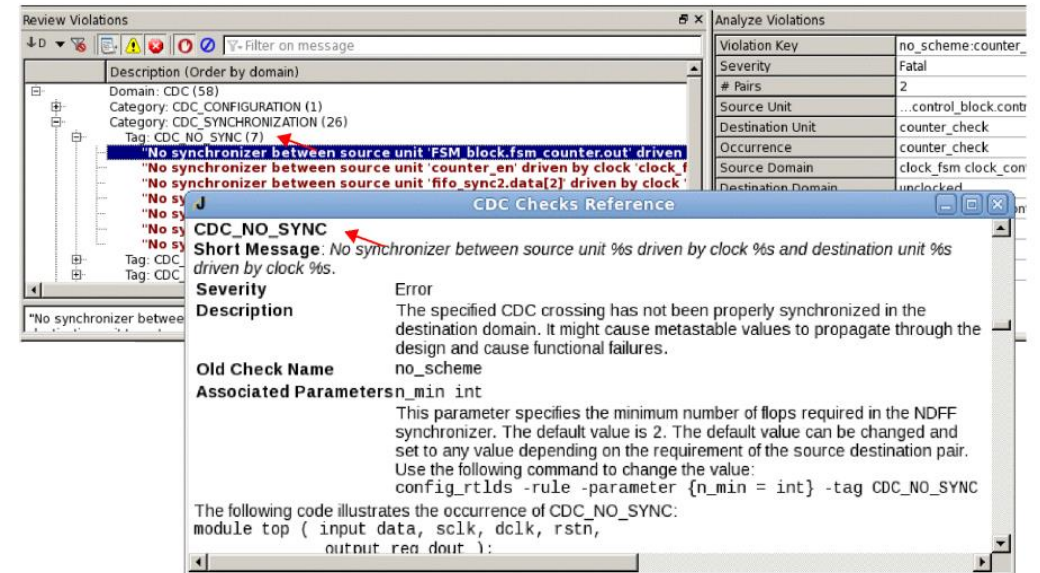


---

# CDC Rules

# FF Synchronizer Rules

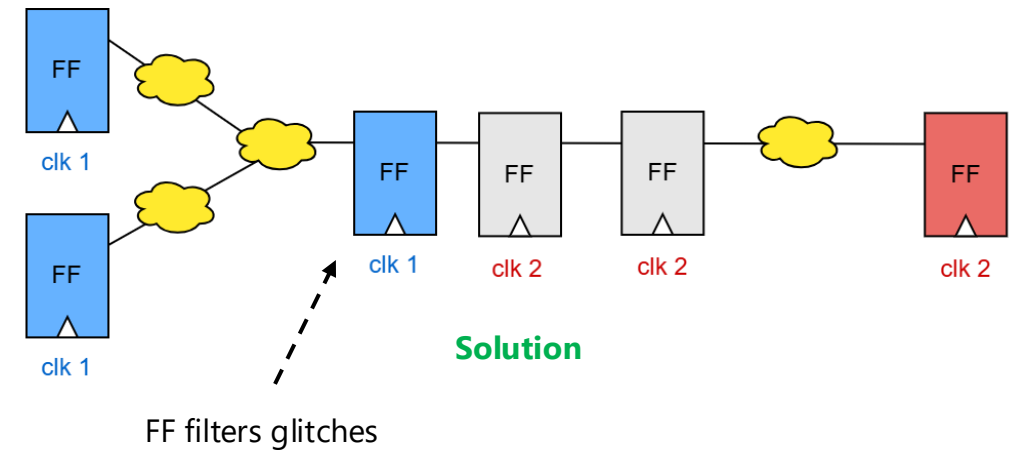
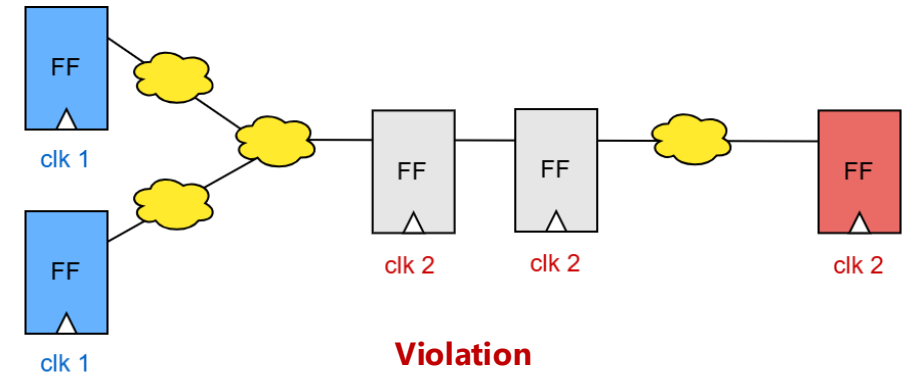
- VLSI CDC tools are used to address issues related to clock domain crossings.
- They can detect if a CDC signal is not synchronized or if it is synchronized but without the proper design practices.
- The tools check and analyze the design RTL and then generate reports addressing the potential issues. The issues reported have different severity levels (warning, critical, etc)
- In the following section we will show some of the CDC rules related to FF synchronizers



Example CDC Report From  
Cadence Jaspergold

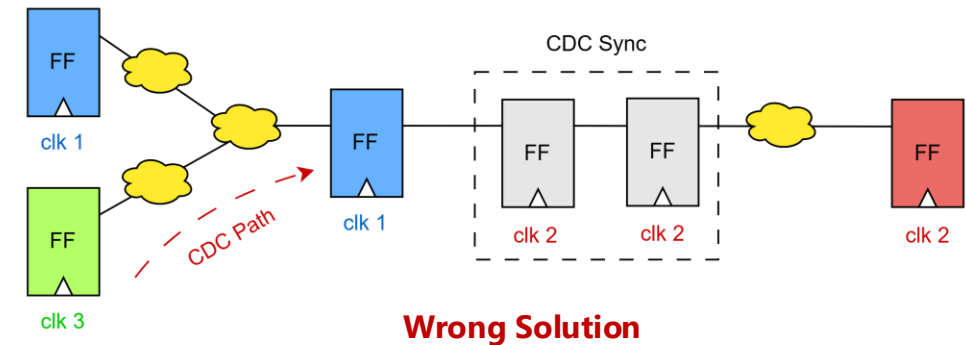
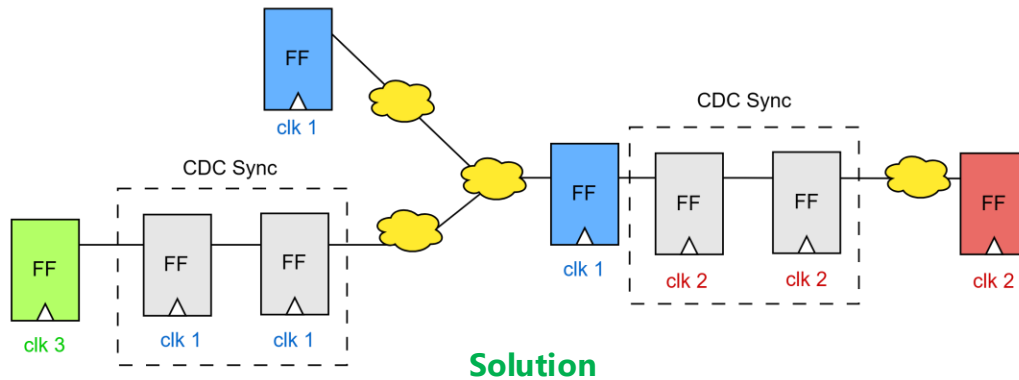
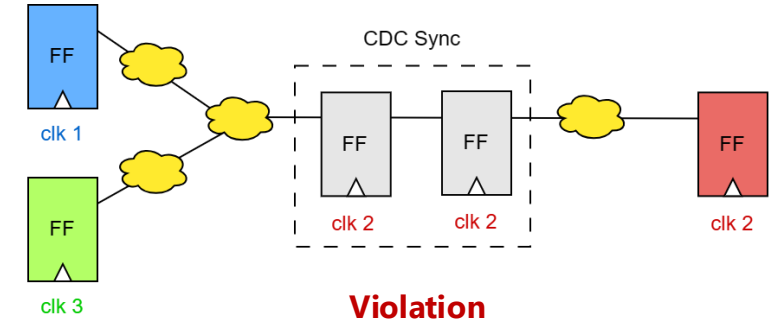
# Convergence in The Sending Domain / Combinational in Sync Fan-in

- Logic converging in the sending domain will cause lots of glitches and we saw how that affects the activity factor and therefore the MTBF.
- Most CDC tools will produce a critical message or even an error if combinational logic was detected in front of the synchronizers.
- **How to fix:**
  - Add a register after the convergence point in the sending domain then pass the clean signal to the CDC synchronizers.



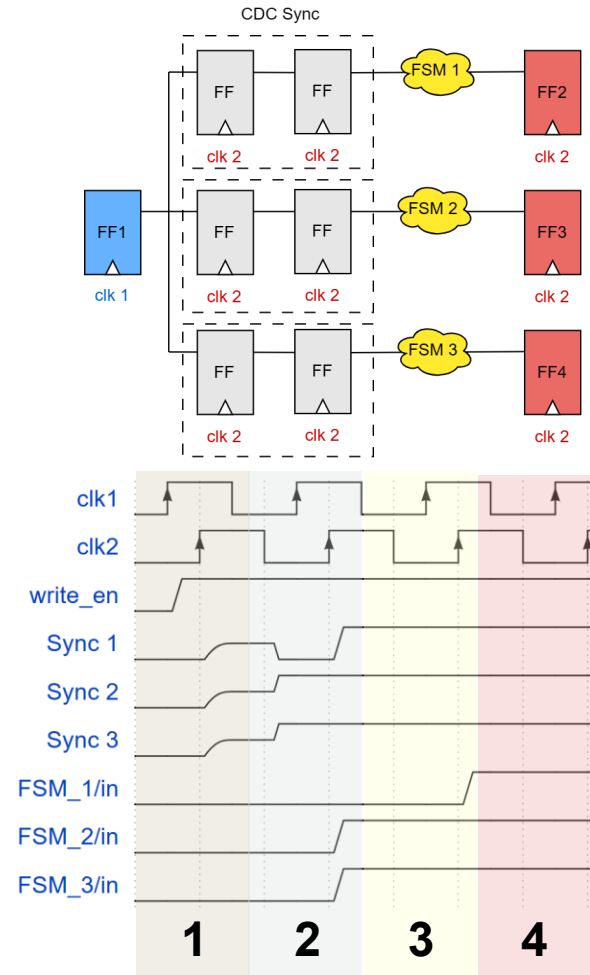
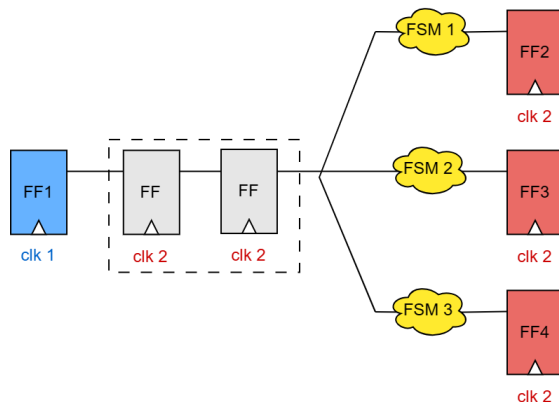
# Multi Clock Fan-in

- We get this violation when signals from multiple clock domains converge and are sent to the CDC sync.
- This will increase the glitches which will damage the MTBF.
- Also, this makes it difficult for CDC tools to properly identify and analyze the design.
- **How to fix:**
  - A wrong solution is to add a glitch filtering FF. because this will lead to a new CDC path as shown.
  - The correct solution is to synchronize the signal from clk\_3 to clk\_1 or from clk\_1 to clk\_3, do the computation in a single domain, then pass it to the CDC sync.



# Divergence in the Sending Domain

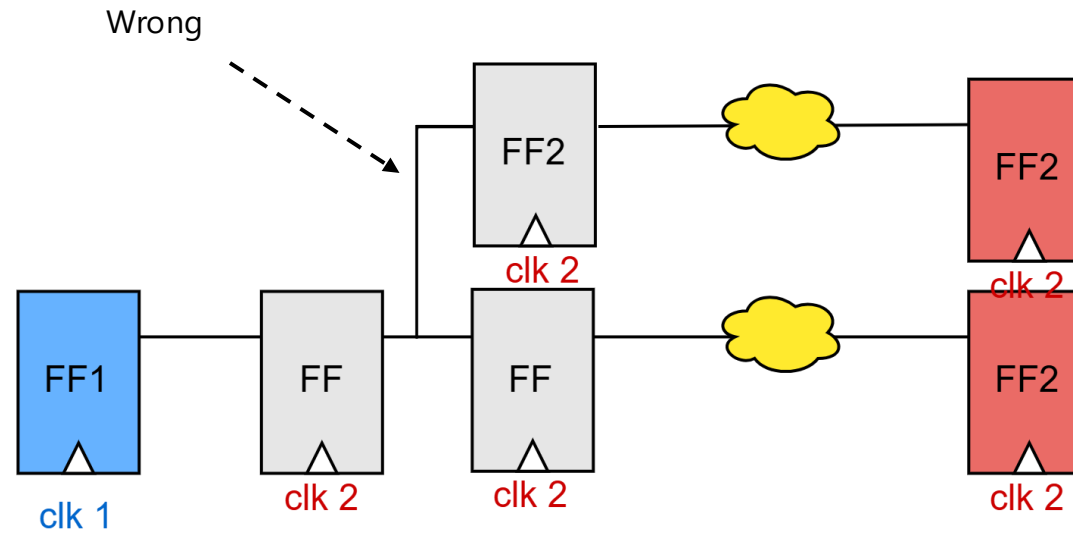
- Divergence occurs when a CDC signal is passed by multiple synchronizers to the other domain.
- **Consider the example on the right:**
  1. Domain 1 sends a write\_enable signal to domain 2 through 3 parallel synchronizers. The change occurs close to the edge of clk2 causing a metastability in all 3 synchronizers.
  2. The 1<sup>st</sup> sync exits metastability and settle at logic 0. The other 2 syncs settle at logic 1.
  3. The FSMs in domain 2 receive the signals from the syncs. The 1<sup>st</sup> FSM doesn't see an active write\_enable signals so it remains IDLE, the other 2 FSMs see an active signal and act accordingly. We have incoherency in the system.
  4. In the next cycle, all FSMs receive the correct value but the damage is already done.
- **How to fix<sup>1</sup>:**
  - Pass the signal with one synchronizer then diverge/fanout at the receiving domain<sup>1</sup>



[1]: The issue might appear silly and rare to happen however it may occur due to lack of good communication among the team members: One engineer create a sync inside their block to pass a global signal to another domain, another engineer create another sync inside their module to pass the same signal. It's better to create a single and separate module for CDC synchronization

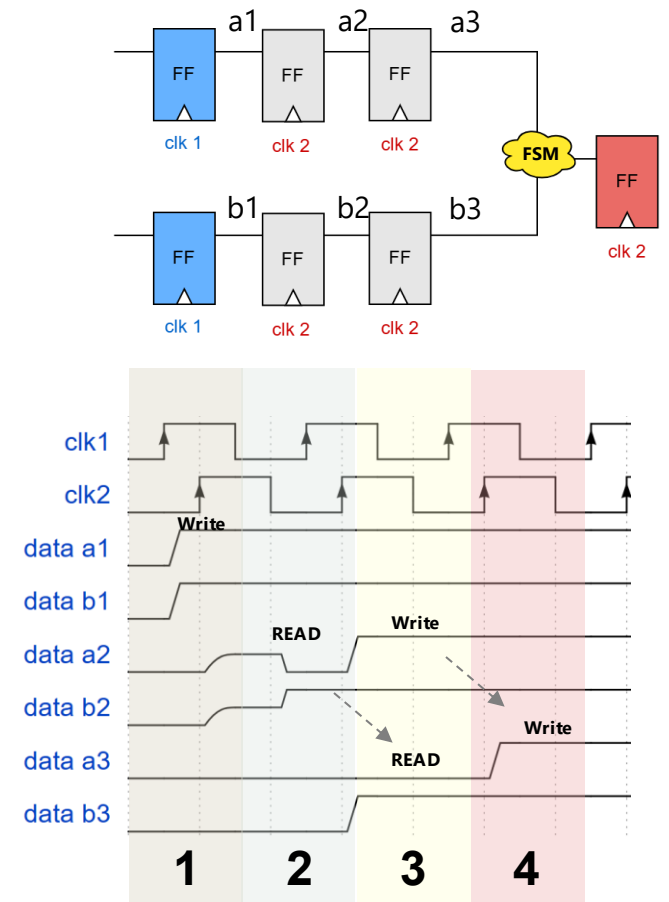
# Divergence of a Metastable Signal

- This issue is similar to the previous one, you shouldn't read/fanout the value between the synchronizers.
- **How to fix:**
  - Pass the signal with one synchronizer then diverge/fanout at the receiving domain



# Multi Signal Reconvergence in the Receiving Domain

- When multiple signals are passed from the sending domain and then converge in the receiving domain, as shown in the diagram, we may get functional errors due to the difference in the settling time between the two signals.
- Consider the example on the right:** Domain 1 sends a 2-bit control signal to domain 2. Initially we are in IDLE=2'b00
  - Domain 1 sends "2'b11 (WRITE)" to domain 2. The change occurs close to the edge of clk2 causing a metastability.
  - Signals a2 and b2 leave metastability and settle at different values "2'b10" (READ).
  - The value "2'b10" (READ) is passed to a3 and b3 and then to the combinational logic causing it to go to a (READ) state while the intended state was (WRITE).
  - The logic receives the correct value (WRITE) later but the damage is already done.
- This example shows the problem with sending multiple signals from one domain to another even with just 2 bits.
- How to fix:**
  - Converge these signals in the sending domain and send them to the receiving domain as one signal. However, this is not always possible.
  - Use gray encoding to make sure only one signal changes at a time (Will be discussed later)
  - Use MUX synchronization scheme to pass these signals as a group (Will be discussed later)



# Conclusion

---

- **We want to know what we solved till now. Our CDC concerns were:**
  - Data corruption: **Partially fixed**. The system, till now, can only send 1-bit data. Also, this data has a varying arrival time.
  - Data incoherence: **Fixed**. The metastable value settles within the synchronizers at 0 or 1 and then propagates to all the 2<sup>nd</sup> domain blocks with the same settled value
  - Data loss: **Fixed**. The pulse is wide enough that it won't be missed by domain 2
  - Data duplication: **Not fixed**
  - Chip burning: **Fixed**. We limited the metastability propagation between the synchronizers and reduced its occurrence frequency.
- In the next parts we will see how to deal with the remaining concerns.



# References

---


- 1) <https://www.uio.no/studier/emner/matnat/ifi/IN3160/v21/timeplan/in3160-l92-clock-domains.pdf>
- 2) <https://ieeexplore.ieee.org/document/1676187>
- 3) <https://www.edn.com/keep-metastability-from-killing-your-digital-design/>
- 4) <https://people.ece.ubc.ca/~edc/7660.jan2018/lec11.pdf>
- 5) <https://www.onsemi.com/pub/Collateral/AN1504-D.PDF>
- 6) <https://www.linkedin.com/in/lukas-vik/recent-activity/articles/>

# Clock Domain Crossing

Part 3

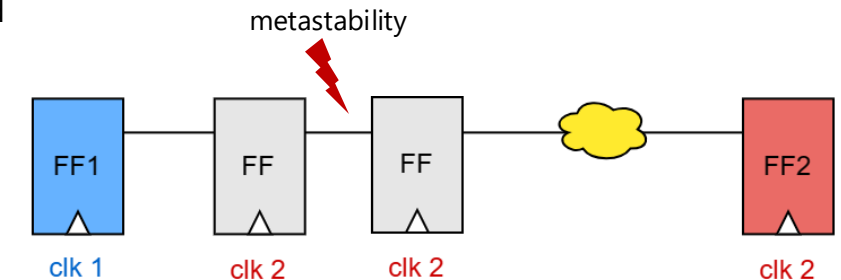
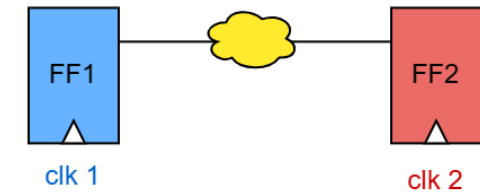
---

Amr Adel Mohammady

 /amradelm

# Introduction

- In the previous part we discussed CDC synchronizers their rules and how to ensure safe capture by launching a wide pulse.
- **Our CDC concerns till now are:**
  - Data corruption: **Partially fixed**. The system, till now, can only send 1-bit data. Also, this data has a varying arrival time.
  - Data incoherence: **Fixed**. The metastable value settles within the synchronizers at 0 or 1 and then propagates to all the 2<sup>nd</sup> domain blocks with the same settled value. We saw the divergence issues and learned how to avoid them.
  - Data loss: **Fixed**. The pulse is wide enough that it won't be missed by domain 2
  - Data duplication: **Not fixed**
  - Chip burning: **Fixed**. We limited the metastability propagation between the synchronizers and reduced its occurrence frequency.
  - In this part we will handle data duplication



---

# Data Duplication

# Data Duplication

- Consider the following example : Domain 1 sends a control signal to domain 2 to enable a counter. Each enable means one count up

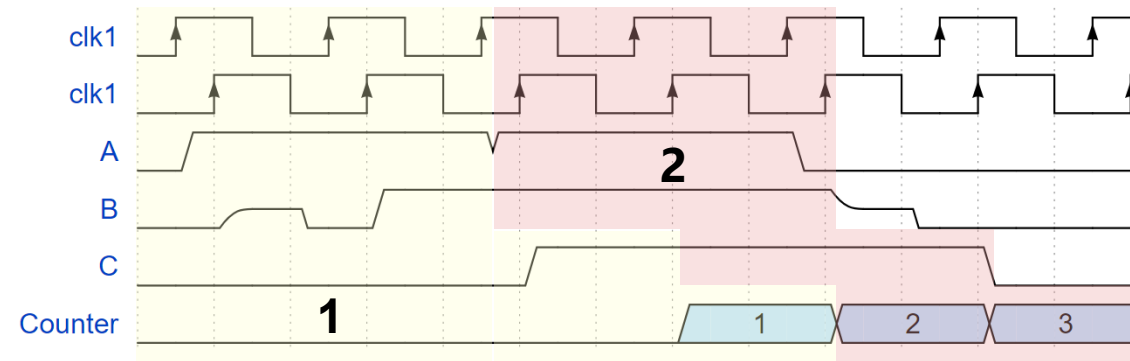
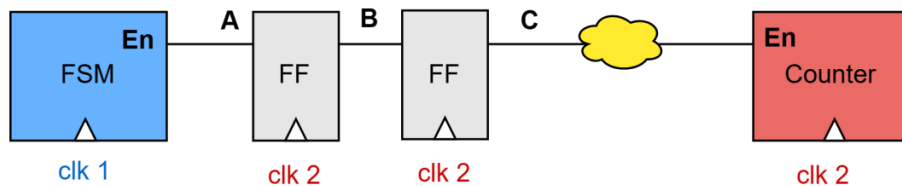
1. Domain 1 sends a logic "1" to domain 2.

- The data goes through metastability and settles at "logic 0".
- Then finally settles at 1 because the pulse was wide enough.
- It then safely reach C and cause the counter to increment one count.

2. Domain 1 sends another logic "1" to domain 2.

- This time the data come out of metastability as "logic 1" which happens to be the correct data.
- The next cycle we get another "logic 1" because we made the pulse wide to ensure safe capture.
- The counter gets 2 enable signals causing it to count 2 times while it was intended to count only once.

- This example shows the problem with data duplication even if the two clocks have the same frequency.
- We need to find a way to make domain 2 sees a single cycle pulse for each wide pulse sent by domain 1



---

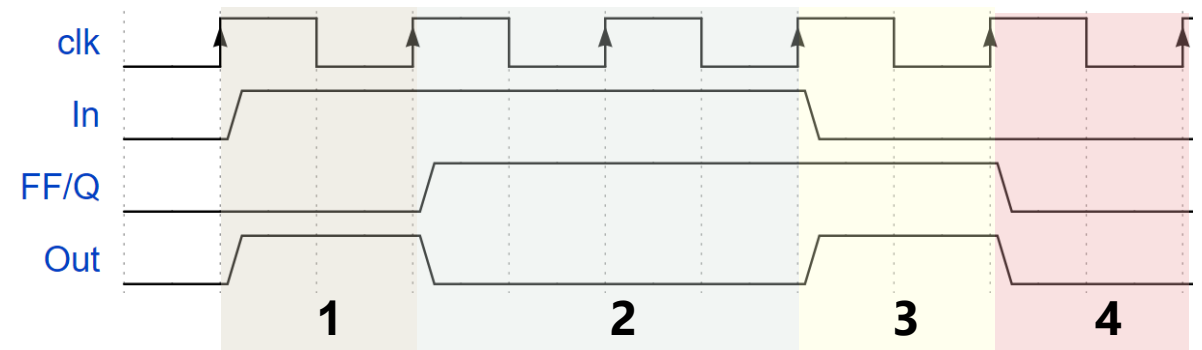
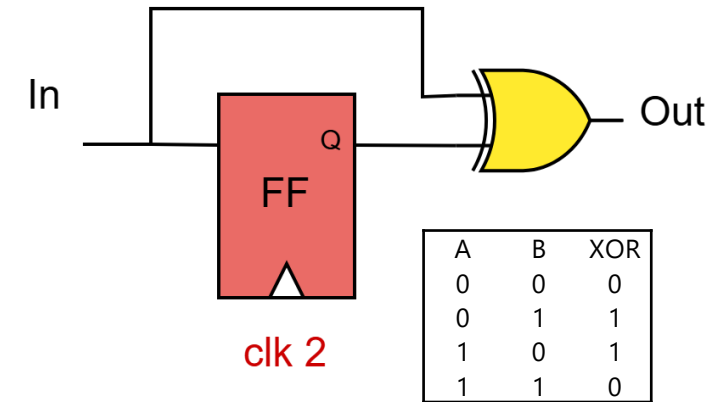
# Pulse Synchronizer

# Pulse Generator

- Consider the circuit on the right:

1. The input **changes** to logic 1.  
The XOR inputs are ( $In = 1$ ) and ( $FF/Q = 0$ ) so the ( $output = 1$ )
2. The input remains at logic 1. The FF now stores the previous value "1".  
The XOR inputs are ( $In = 1$ ) and ( $FF/Q = 1$ ) so the ( $output = 0$ )  
The output remains "0" as long as the input remains constant.
3. The input **changes** to logic 0.  
The XOR inputs are ( $In = 0$ ) and ( $FF/Q = 1$ ) and so the ( $output = 1$ )
4. The input remains at logic 0. The FF now stores the previous value "0".  
The XOR inputs are ( $In = 0$ ) and ( $FF/Q = 0$ ) so the ( $output = 0$ )  
The output remains "0" as long as the input remains constant.

- As we can see, this circuit outputs "1" for one cycle whenever the input **changes/toggles**. And outputs "0" as long as the input remain constant.
- This circuit is called a pulse generator and will help us deal with the CDC data duplication issue.

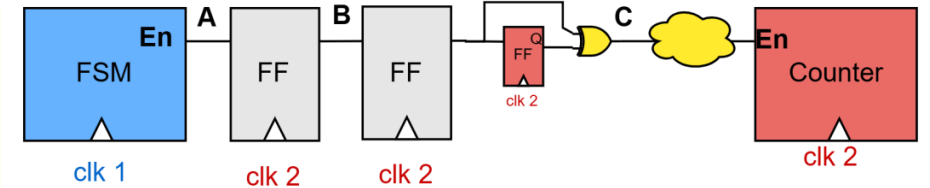


# Data Duplication

- Let's analyze the same example we saw earlier but now with the pulse generator added:

1. Domain 1 sends a logic "1" to domain 2.

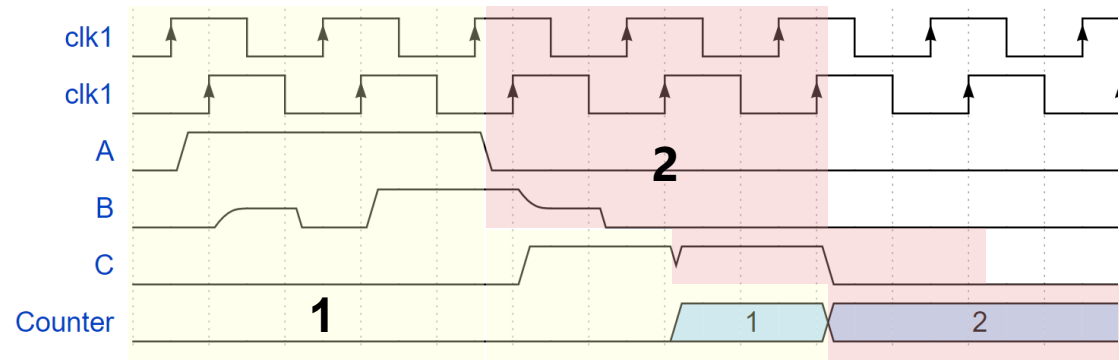
- The data goes through metastability and settles at "logic 0".
- Then finally settles at 1 because the pulse was wide enough.
- It then safely reaches the pulse generator. The pulse generator ensures that the pulse remains 1 for only one cycle as long as the input remains constant



2. After some time domain 1 sends another enable to domain 2. The enable is sent by toggling the signal to "0".

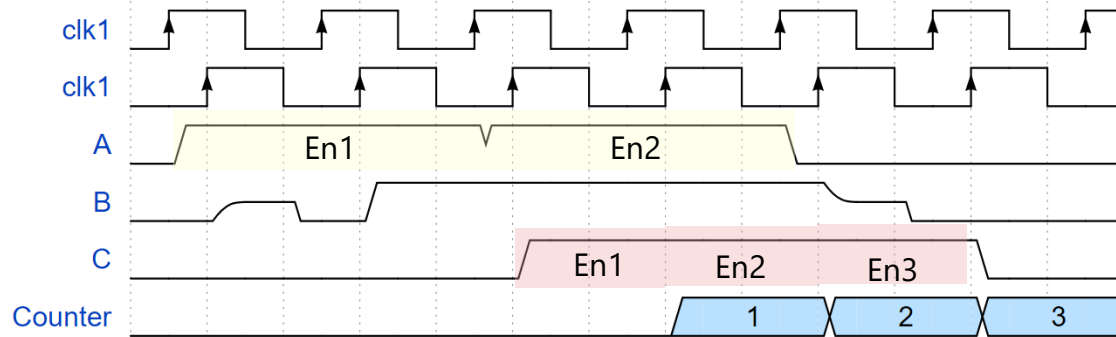
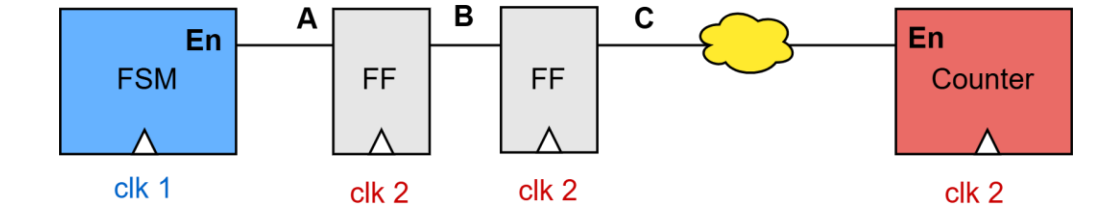
- The data goes through metastability and settles at "logic 0".
- The next cycle we get another "logic 0" because we made the pulse wide to ensure safe capture.
- The pulse generator outputs "logic 1" for only one cycle.

- As you can see the pulse generator fixed the data duplication issue because it converts a wide pulse into a single cycle pulse.

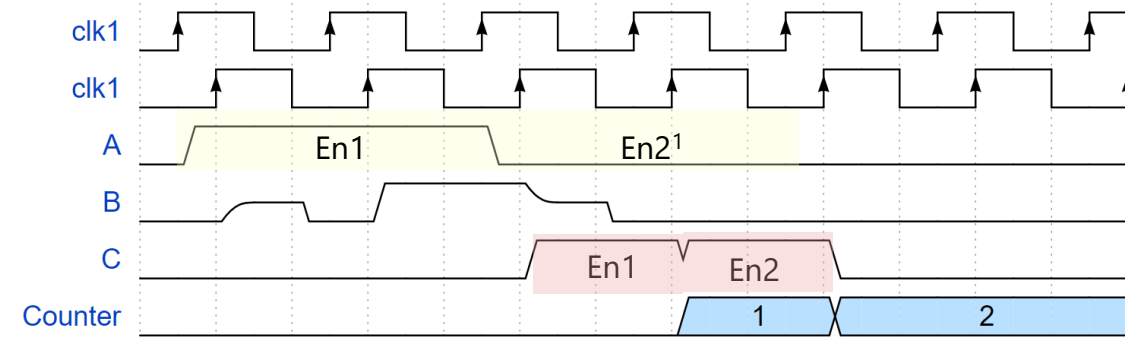
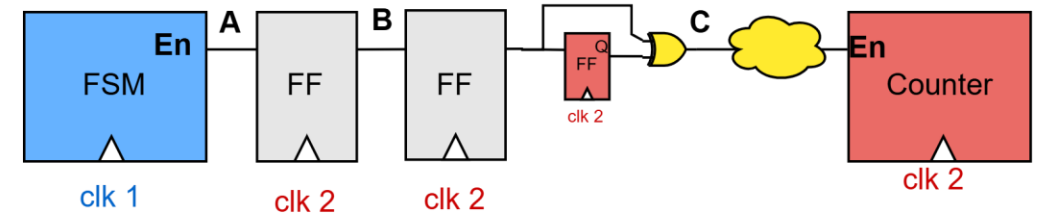




# Data Duplication



**Without Pulse generator**



**With Pulse generator**

[1]: Notice how En2 in the pulse generator example is a toggle and not logic 1

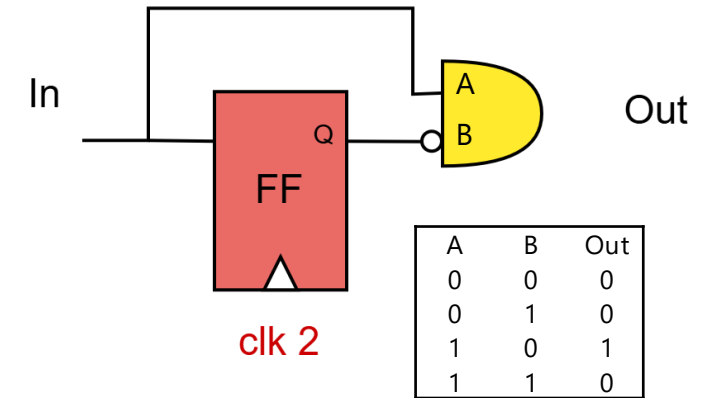
---

# Edge Synchronizers

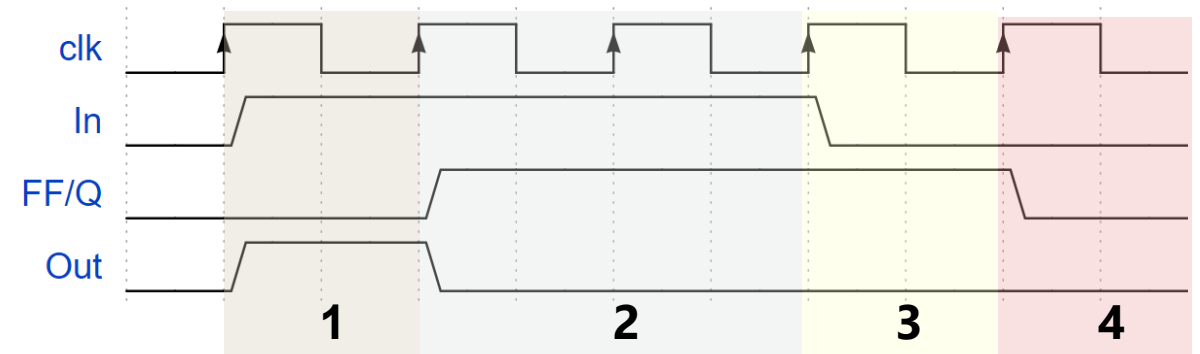
# Edge Detector

- Consider the circuit on the right:

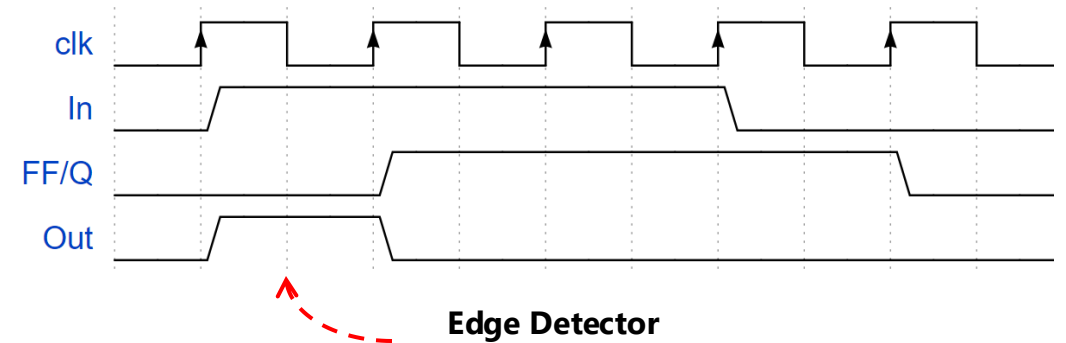
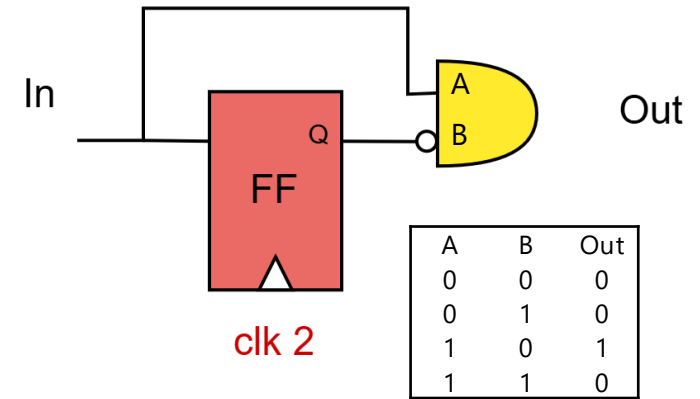
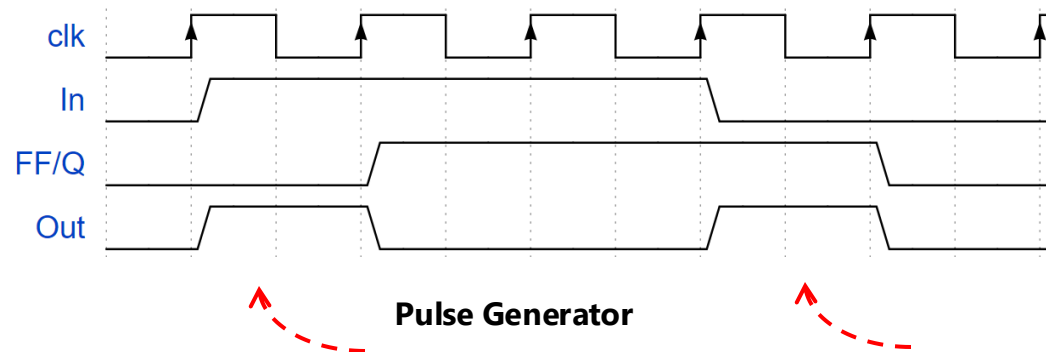
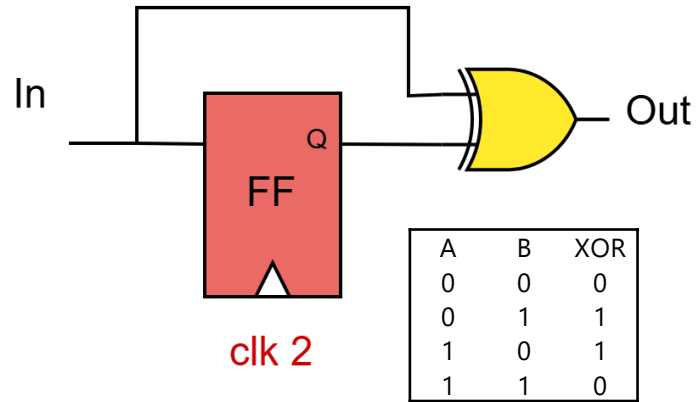
1. The input changes to logic 1.  
The inputs to the AND are In: "1" and FF/Q: "0" so the output: "1"
2. The input remains at logic 1. The FF now stores the previous value "1".  
The AND inputs are In: "1" and FF/Q: "1" so the output: "0"  
The output remains "0" as long as the input remains constant.
3. The input changes to logic 0.  
The inputs to the AND are In: "0" and FF/Q: "1" and so the output: "0"
4. The input remains at logic 0. The FF now stores the previous value "0".  
The AND inputs are In: "0" and FF/Q: "0" so the output: "0"  
The output remains "0" as long as the input remains constant.



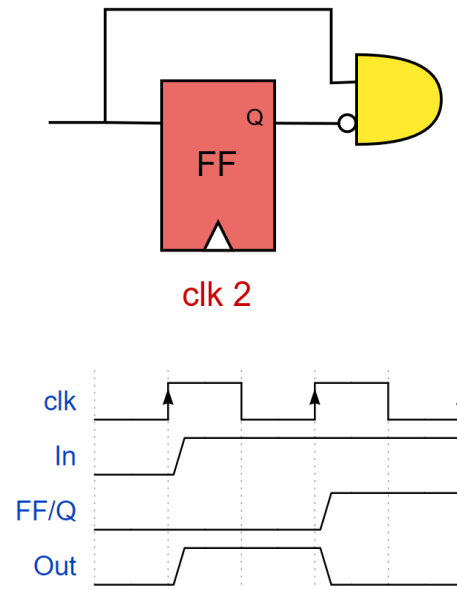
- Compared to the pulse generator, this circuit produced a pulse only when the input changed from 0 → 1 while the pulse generator produced a pulse whether the input changed from 0 → 1 or from 1 → 0.



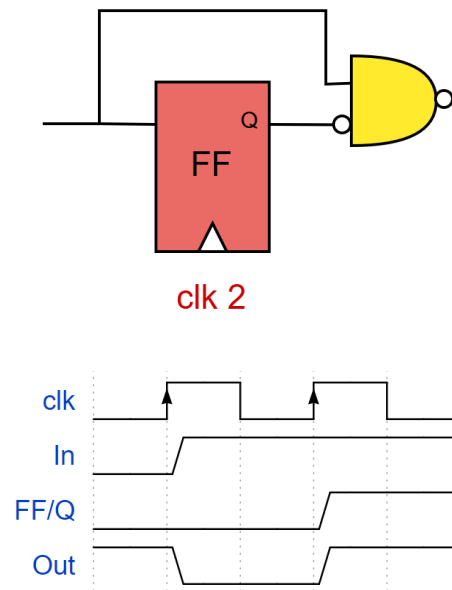
# Pulse Generator vs Edge Detector



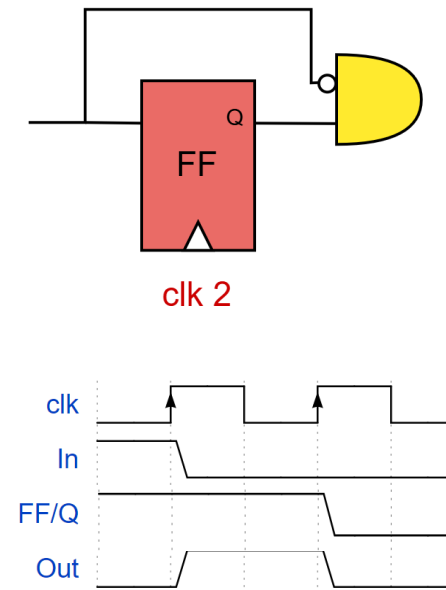
# Edge Detector Types



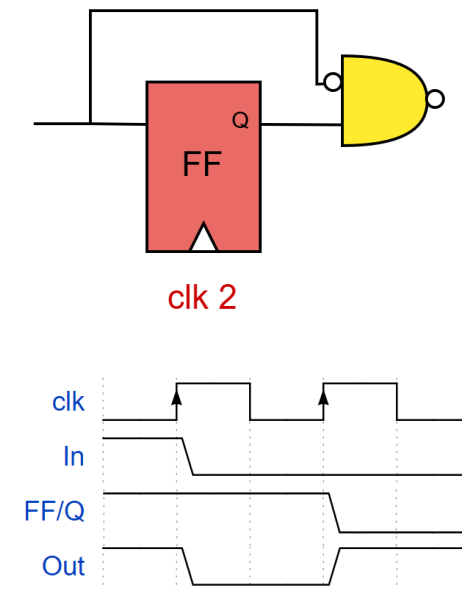
**Rising Edge Detection  
Active High Output**



**Rising Edge Detection  
Active Low Output**



**Falling Edge Detection  
Active High Output**



**Falling Edge Detection  
Active Low Output**

# Conclusion

---

- **We want to know what we solved till now. Our CDC concerns were:**
  - Data corruption: **Partially fixed**. The system, till now, can only send 1-bit data. Also, this data has a varying arrival time.
  - Data incoherence: **Fixed**. The metastable value settles within the synchronizers at 0 or 1 and then propagates to all the 2<sup>nd</sup> domain blocks with the same settled value
  - Data loss: **Fixed**. The pulse is wide enough that it won't be missed by domain 2
  - Data duplication: **Fixed**
  - Chip burning: **Fixed**. We limited the metastability propagation between the synchronizers and reduced its occurrence frequency.
- In the next parts we will see how to send a multi-bit signal.

# References

---


- 1) <https://www.uio.no/studier/emner/matnat/ifi/IN3160/v21/timeplan/in3160-l92-clock-domains.pdf>
- 2) <https://ieeexplore.ieee.org/document/1676187>
- 3) <https://www.edn.com/keep-metastability-from-killing-your-digital-design/>
- 4) <https://people.ece.ubc.ca/~edc/7660.jan2018/lec11.pdf>
- 5) <https://www.onsemi.com/pub/Collateral/AN1504-D.PDF>

# Clock Domain Crossing

Part 4

---

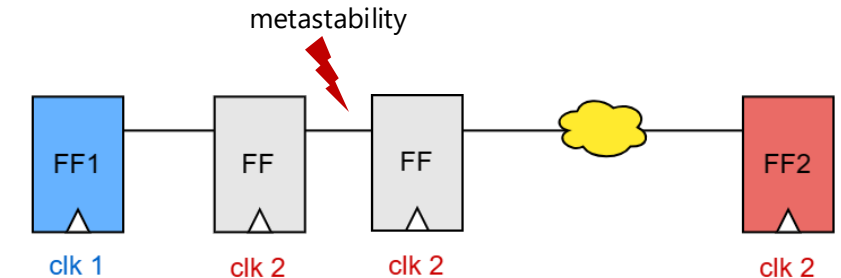
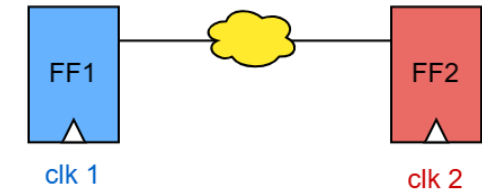
Amr Adel Mohammady

 /amradelm



# Introduction

- In the previous parts we saw how to send single bit across different clock domains.
- We also saw how to handle data duplication using pulse and edge synchronizers
- **Our CDC concerns till now:**
  - Data corruption: **Partially fixed**. The system, till now, can only send 1-bit data. Also, this data has a varying arrival time.
  - Data incoherence: **Fixed**. The metastable value settles within the synchronizers at 0 or 1 and then propagates to all the domain 2 blocks with the same settled value
  - Data loss: **Fixed**. The pulse is wide enough that it won't be missed by domain 2
  - Data duplication: **Fixed**. slf we use pulse/edge synchronizers
  - Chip burning: **Fixed**. We limited the metastability propagation between the synchronizers and reduced its occurrence frequency.
- In this part we will see how to send multi-bit signal using gray coding

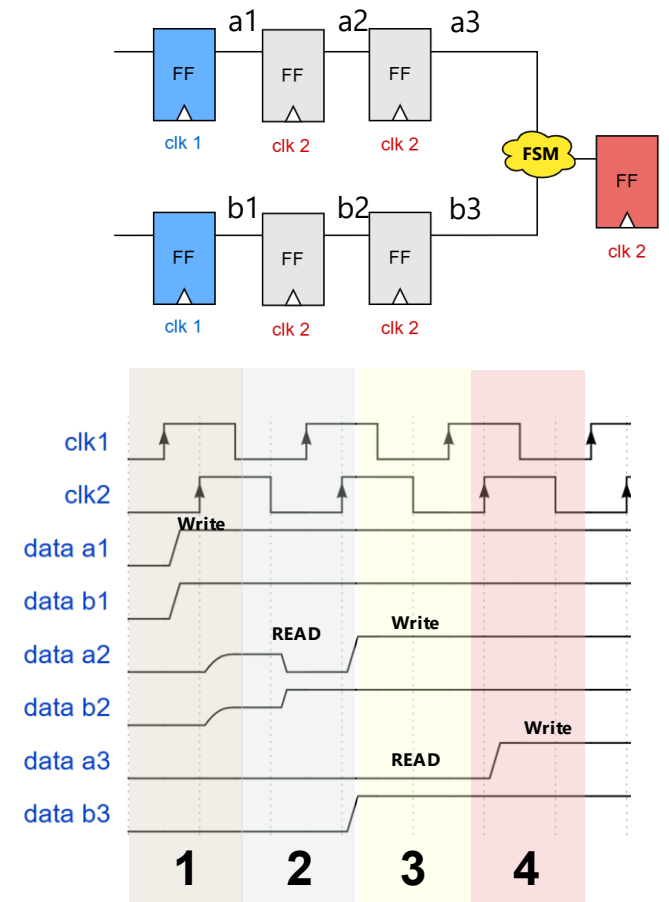


---

# Reconvergence Issue

# Multi Signal Reconvergence in the Receiving Domain

- When multiple signals are passed from the sending domain and then converge in the receiving domain, as shown in the diagram, we may get functional errors due to the difference in the settling time between the two signals.
- Consider the example on the right:** Domain 1 sends a 2-bit control signal to domain 2. Initially we are in IDLE=2'b00
  - Domain 1 sends "2'b11 (WRITE)" to domain 2. The change occurs close to the edge of clk2 causing a metastability.
  - Signals a2 and b2 leave metastability and settle at different values "2'b10" (READ).
  - The value "2'b10" (READ) is passed to a3 and b3 and then to the combinational logic causing it to go to a (READ) state while the intended state was (WRITE).
  - The logic receives the correct value (WRITE) later but the damage is already done.
- This example shows the problem with sending multiple signals from one domain to another even with just 2 bits.
- How to solve:**
  - Converge these signals in the sending domain then send them as one signal to the receiving domain. However, this is not always possible.
  - Use gray encoding to make sure only one signal changes at a time
  - Use MUX synchronization scheme to pass these signals as a group (Will be discussed later)

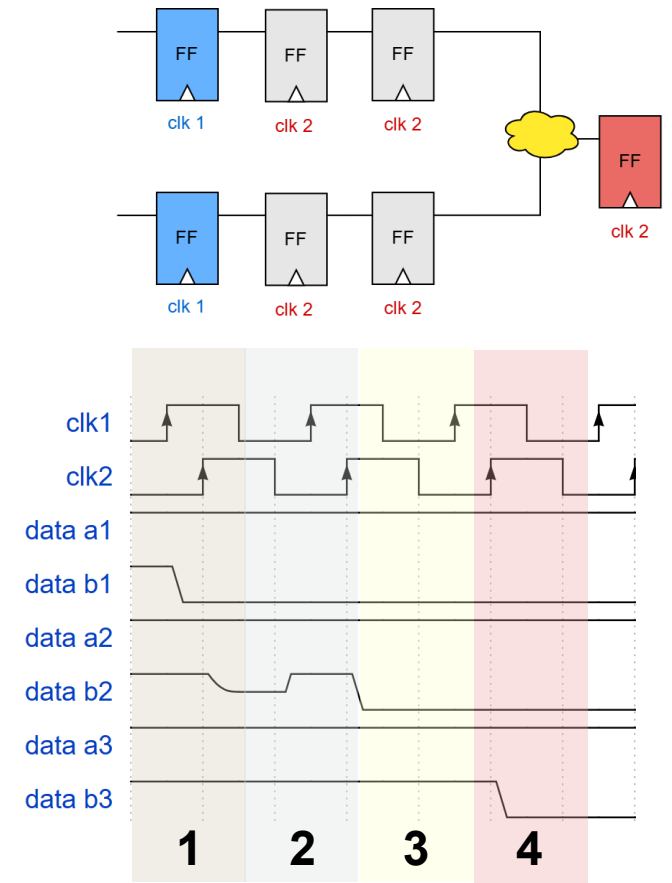
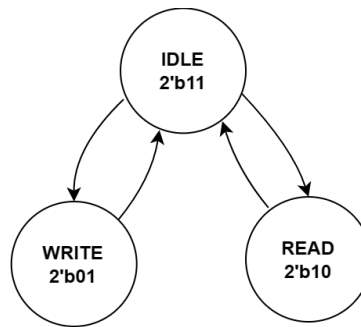


---

# Gray Coding

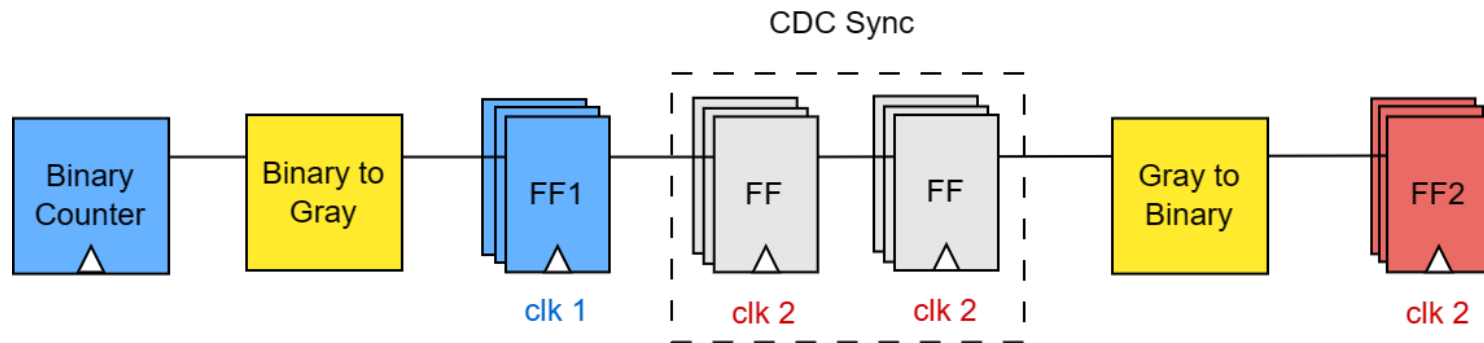
# Gray Coding

- In the previous example we saw how sending multi signals across clock domains can be problematic.
  - If we can ensure only one bit changes at a time we can solve the issue because we will either get the current value or the new value.
  - Lets see an example to understand:** As the previous example, we want to send a control signal from domain 1 to domain 2. The only difference is we will change the bit assignments. The diagram below shows the possible state transitions. Initially we are in IDLE=2'b11
- Domain 1 sends "2'b01 (WRITE)" to domain 2. The change occurs close to the edge of clk2 causing a metastability. This time, only signals b1 changed while a1 remained stable hence a2 won't go metastable
  - Signals b2 leave metastability and settle at wrong value "1'b1".
  - The value "2'b11" (IDLE) is passed to a3 and b3 and then to the combinational logic. The logic still sees (IDLE) so nothing changes in the system.
  - The logic receives the correct value "2'b01" (WRITE) later and the system enters the correct state.
- Because we ensured only 1 bit changes at a time, domain 2 will either see the old value (IDLE) or the new value (WRITE). It won't erroneously enter the (READ) state.



# Gray Coding

- Gray code is a binary numbering system where every two **adjacent** values differ only in one bit.
- These codes are widely used with CDC to avoid functional errors and ensure the receiving domain either get the old value or the new value and not any other illegal value as we saw in the previous example.
- One of the main uses of gray codes is passing counter values from one domain to another. This will come in handy soon.**
  - The sending domain has a normal binary counter. The counter output goes to a binary to gray encoder.
  - The output is then sent to FFs to protect the synchronizers from combinational glitches.
  - The encoded values pass through the synchronizer then to the gray to binary decoder.
  - Because only one bit changes we ensure the receiving domain will either receive the old count or the new count.

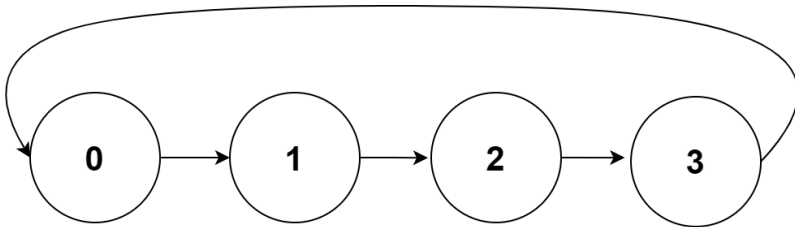


**Passing Counter Values From one Domain to Another**

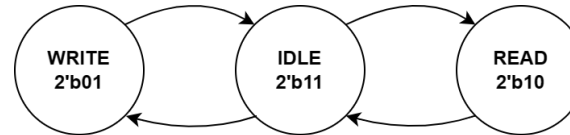
Decimal	Binary Code	Gray Code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111

# Gray Coding

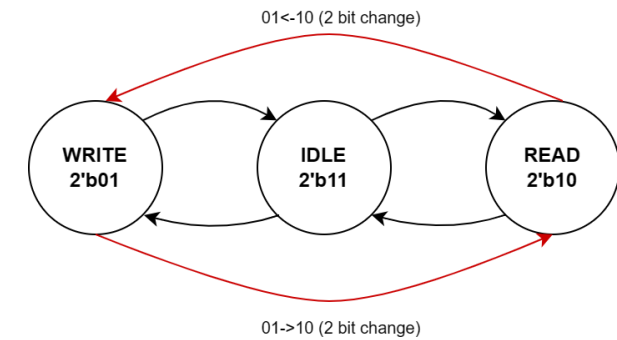
- Using gray codes we managed to send counter values, FSM states or any signal where the transitions are sequential and predictable.
- However, Gray codes can't be used to send data signals or any signal where the transitions are not sequential. For example, if we are sending decimal value such as "562" we have no idea what the next sent data will be and we have no ability to ensure the transition to the new value is done with one bit change.
- In the next part we will discuss how to send data signals or any multi-bit signal that can't be gray coded.



Counter FSM



FSM That **Can** Be Gray Coded




FSM That **Can't** Be Gray Coded

# Clock Domain Crossing

Part 5

---

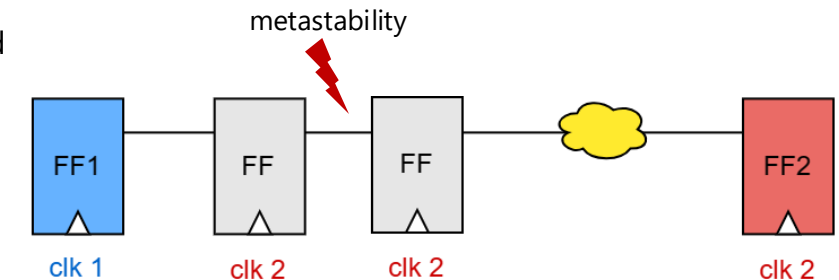
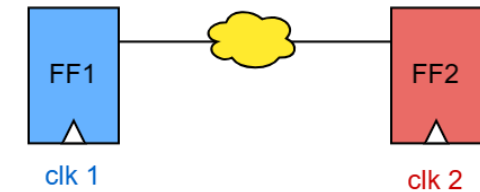
Amr Adel Mohammady

 /amradelm



# Introduction

- In the previous parts we saw how to send single bit across different clock domains.
- We also saw how to handle data duplication using pulse and edge synchronizers
- We then saw how to send multi bit signals using gray coding but that wasn't enough to send data values.
- **Our CDC concerns till now:**
  - Data corruption: **Partially fixed**. The system, till now, can only send multi-bit control signals but not data. Also, this data has a varying arrival time.
  - Data incoherence: **Fixed**. The metastable value settles within the synchronizers at 0 or 1 and then propagates to all the domain 2 blocks with the same settled value
  - Data loss: **Fixed**. The pulse is wide enough that it won't be missed by domain 2
  - Data duplication: **Fixed** if we use pulse/edge synchronizers
  - Chip burning: **Fixed**. We limited the metastability propagation between the synchronizers and reduced its occurrence frequency.
- In this part we will see how to send multi-bit data signal using the mux synchronization scheme

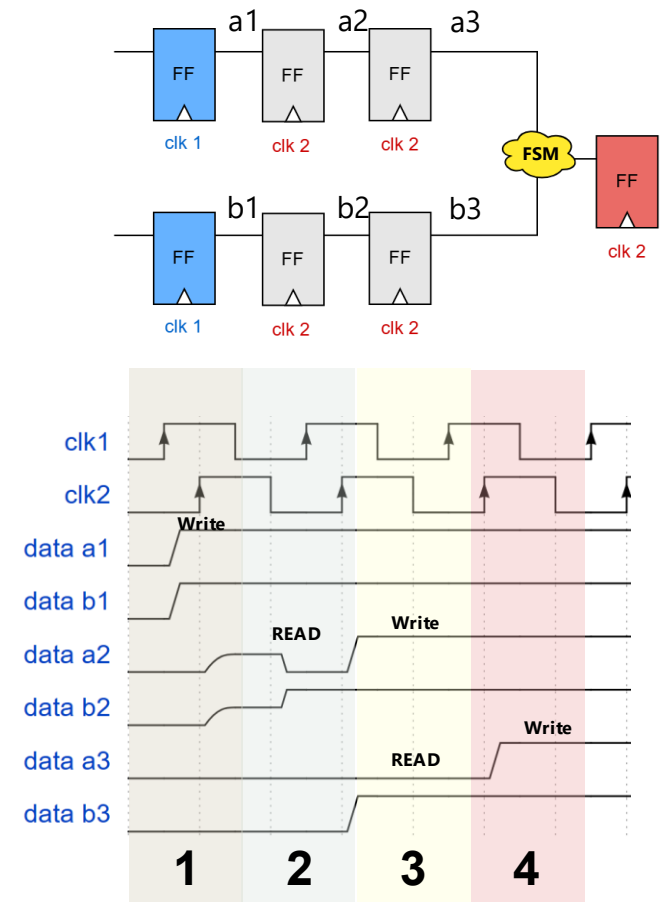


---

# Reconvergence Issue

# Multi Signal Reconvergence in the Receiving Domain

- When multiple signals are passed from the sending domain and then converge in the receiving domain, as shown in the diagram, we may get functional errors due to the difference in the settling time between the two signals.
- Consider the example on the right:** Domain 1 sends a 2-bit control signal to domain 2. Initially we are in IDLE=2'b00
  - Domain 1 sends "2'b11 (WRITE)" to domain 2. The change occurs close to the edge of clk2 causing a metastability.
  - Signals a2 and b2 leave metastability and settle at different values "2'b10" (READ).
  - The value "2'b10" (READ) is passed to a3 and b3 and then to the combinational logic causing it to go to a (READ) state while the intended state was (WRITE).
  - The logic receives the correct value (WRITE) later but the damage is already done.
- This example shows the problem with sending multiple signals from one domain to another even with just 2 bits.
- How to solve:**
  - Converge these signals in the sending domain then send them as one signal to the receiving domain. However, this is not always possible.
  - Use gray encoding to make sure only one signal changes at a time (Discussed before)
  - Use MUX synchronization scheme to pass these signals as a group (Will be discussed now)

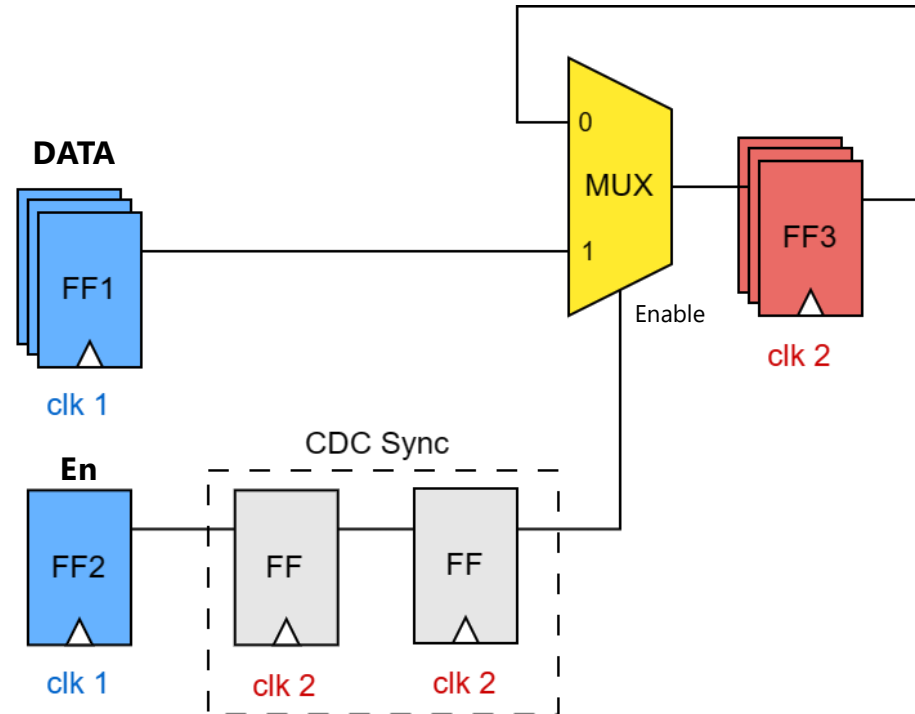


---

# MUX Synchronizer

# MUX Synchronizer

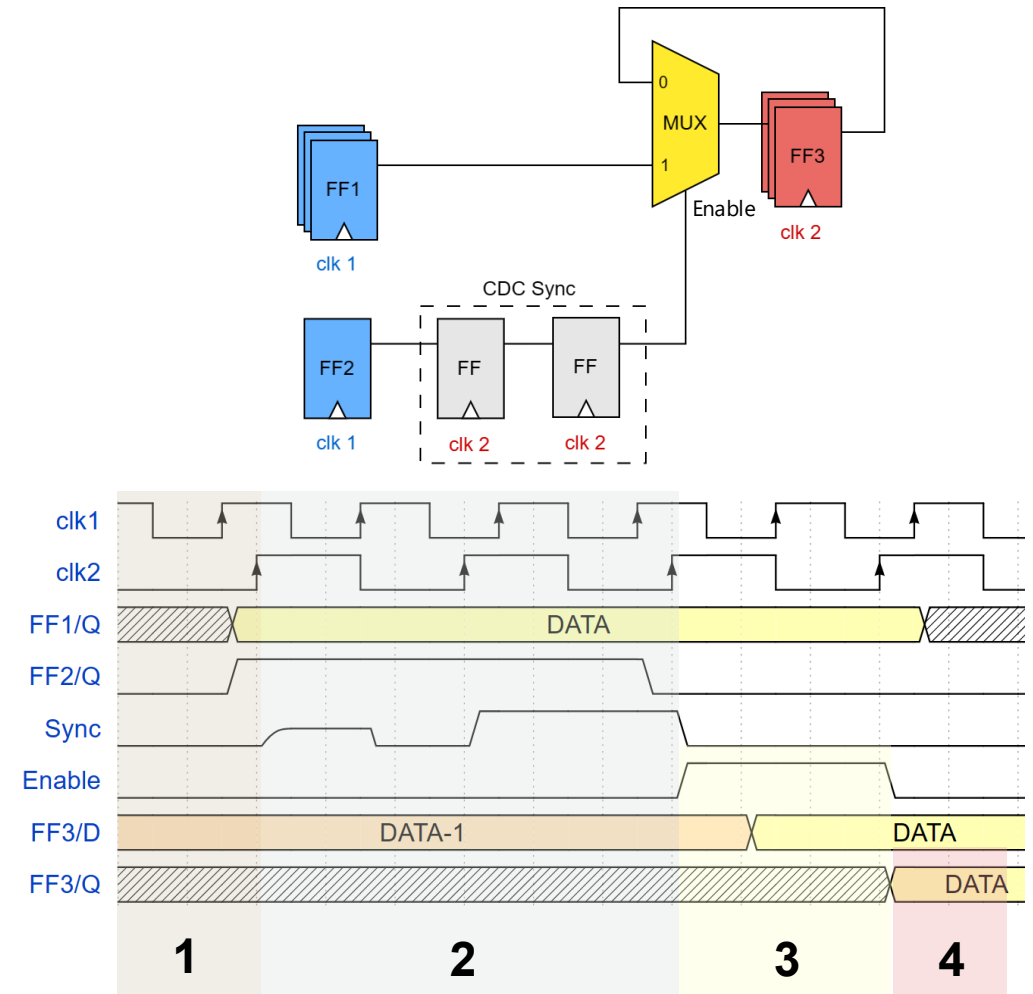
- The MUX synchronizer is one of the methods to pass multiple signals across clock domains.
- **It consists of two parts:**
  - The data part that goes directly to domain 2 without passing through synchronizers
  - An enable signal that pass through synchronizers then to domain 2 and is used to enable domain 2 to read the data of domain 1



# MUX Synchronizer

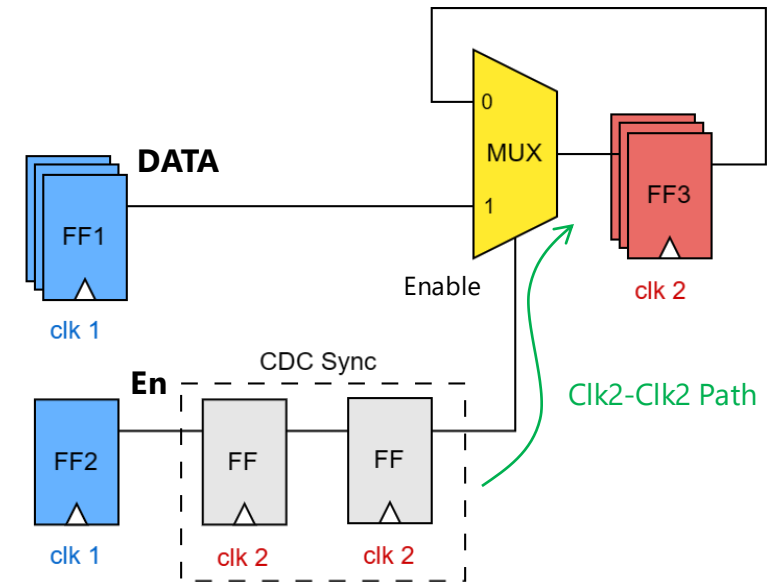
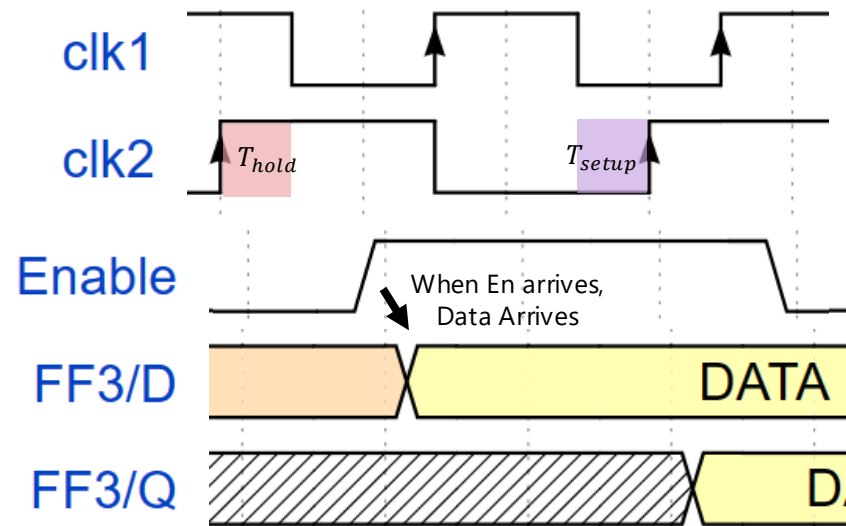
- **The MUX Synchronizer works as follows:**

1. Domain 1 launches both the data and the enable signal to domain 2. The data goes directly to the MUX while the enable signal goes through the synchronizers
2. -The data reaches the MUX and waits for the select to be "1" to reach FF3.  
  
-The MUX in front of FF3 still has select = "0", so the D pin sees the old stored value inside FF3 and therefore FF3 won't go metastable.  
  
-At the same time the enable signals goes through the syncs and cause metastability
3. -After some time the enable signal reaches domain 2 and then the MUX.  
  
-The data now propagates through the MUX to FF3/D.  
  
-This propagation is controlled by the enable signal which is a synchronous signal within domain 2. This signal can be analyzed in STA to make sure FF3/D won't change near a clock edge and won't cause setup or hold violations.
4. The data is captured by FF3



# Why Does It Work?

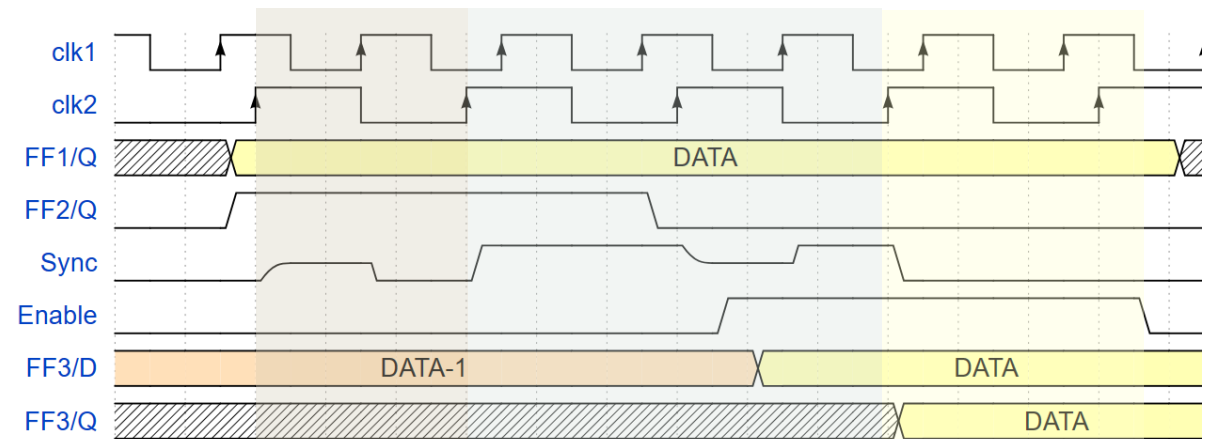
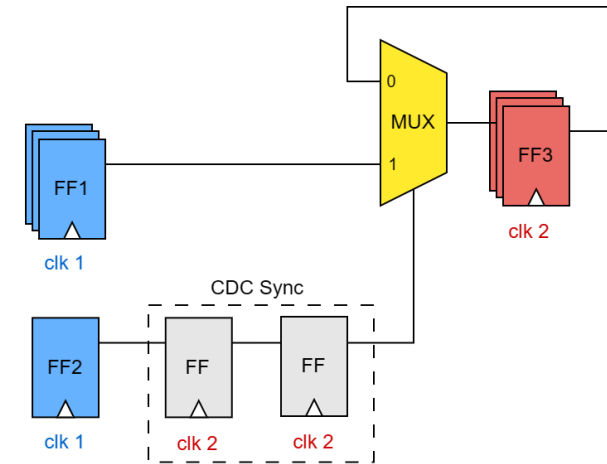
- We are concerned that data from FF1 might reach FF3 during its setup/hold window, potentially causing metastability.
- **Lets ask: Will this happen under this structure?**
  - The enable signal acts as a gate for the data, preventing it from reaching FF3 until the enable signal arrives.
  - The enable signal is a timing path from clk2 to clk2, which can be analyzed using Static Timing Analysis (STA).
  - STA ensures that the enable signal does not arrive (the gate doesn't open) during the metastability window.
  - Therefore also the data will not arrive (pass the gate) during the metastability window.
- This needs an important assumption<sup>1</sup>: That by the time the enable reaches the MUX, the data should've arrived and is stable in front of the MUX



[1]: This assumption need to be enforced with a max delay constraint

# MUX Synchronizer – Data Pulse Width

- For how long do we need to hold data stable to ensure safe capture by the receiving domain?
  - Most resources don't discuss this. Very few mention it's  $N+1$  clk2 cycles where  $N$  is the number of sync FFs, but this can be shown to risk metastability<sup>1</sup>. The analysis below is my own and you are advised to take it with a grain of salt.
  - We have two concerns:
    1. We don't want the data to change before the enable signal arrives. Otherwise, the data won't be captured by domain 2 (The assumption from the previous slide)
    2. We don't want the data to change as long as the enable signal is high. Otherwise, the MUX will pass any asynchronous event from domain 1 to domain 2 and cause metastability.
  - To handle these concerns we need to consider the worst case:
    1. In the first clk2 cycle, the synchronizers goes metastable and settle at "0" instead of "1".
    2. After two ( $N$ ) clk2 cycles, the enable becomes "1". At the same time, domain 1 lowers the enable signal, the change causes metastability and the synchronizers settle at "1" instead of "0".
    3. After one clk2 cycle + the delay from the sync to the MUX, the enable goes low "0". Any change in FF1/Q won't reach FF3/D
- We can assume<sup>2</sup> the sync->MUX delay = 0.5 clk2 cycle.  
So If we have  $N$  number of synchronizers then the data must be held for  $N+2.5$  clk2 cycles



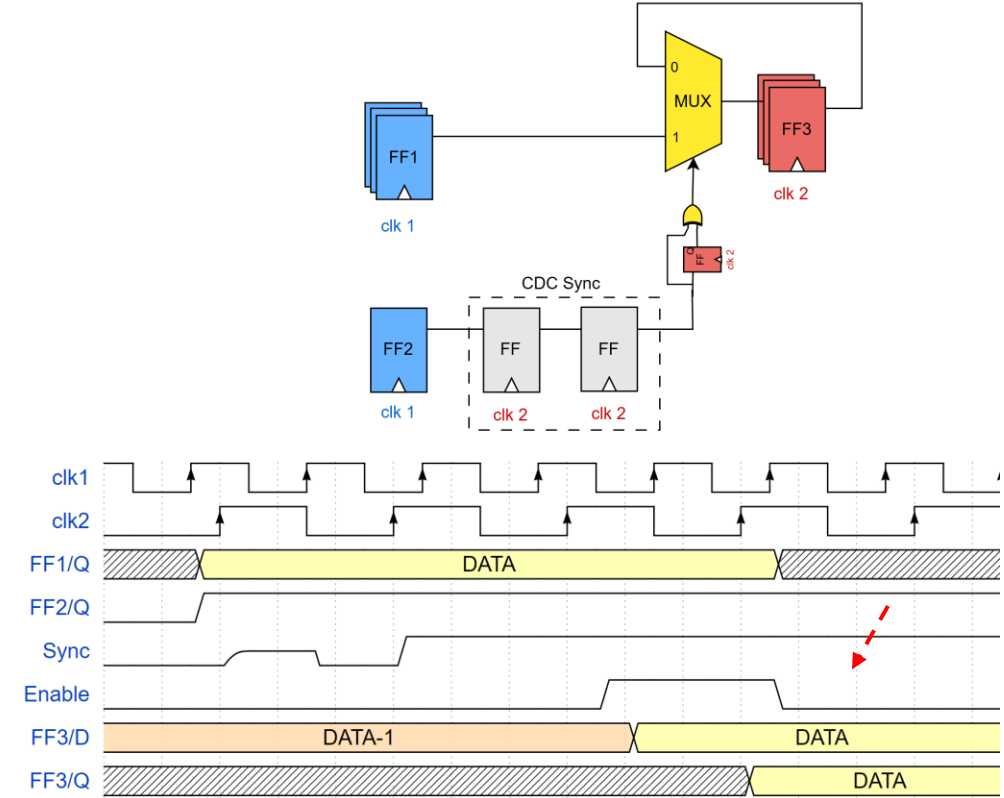
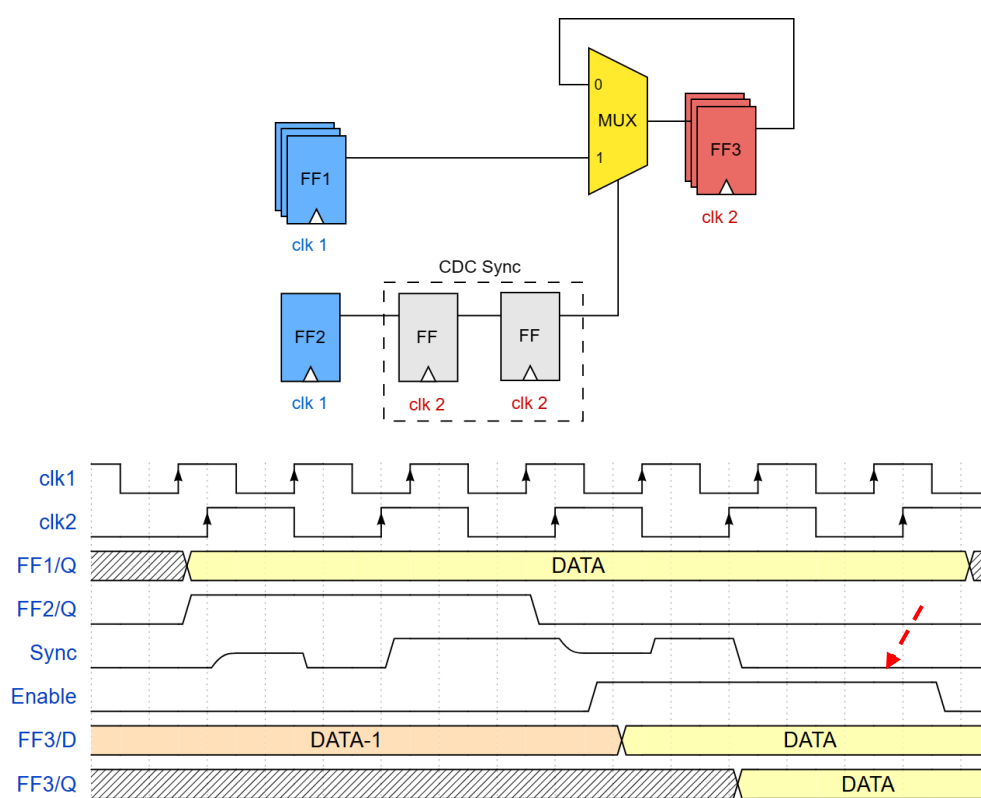
[1] :  $N+1$  will work if we use pulse synchronizers. See next slide

[2] : This assumption need to be enforced with a max delay constraint



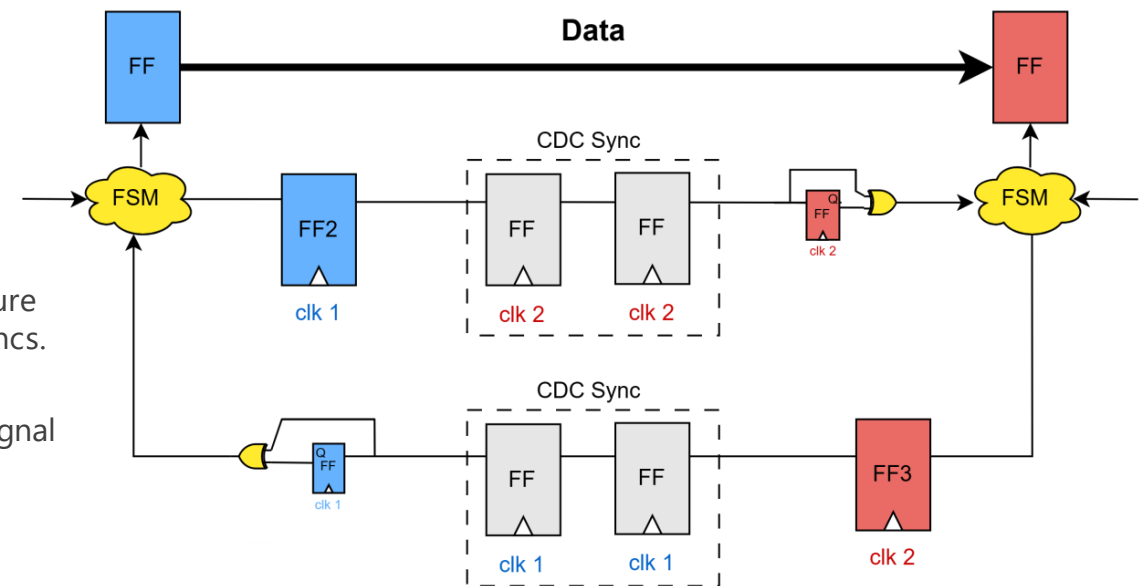
# MUX Synchronizer With Pulse Synchronizer – Data Pulse Width

- Using a pulse synchronizer can shorten the requirement to  $N + 1.5$  cycles instead of  $N + 2.5$  cycles.
- This is because the pulse synchronizer will cause the enable to be active for one cycle instead of 2 (see red arrows).



# Mux Synchronizers - Handshake

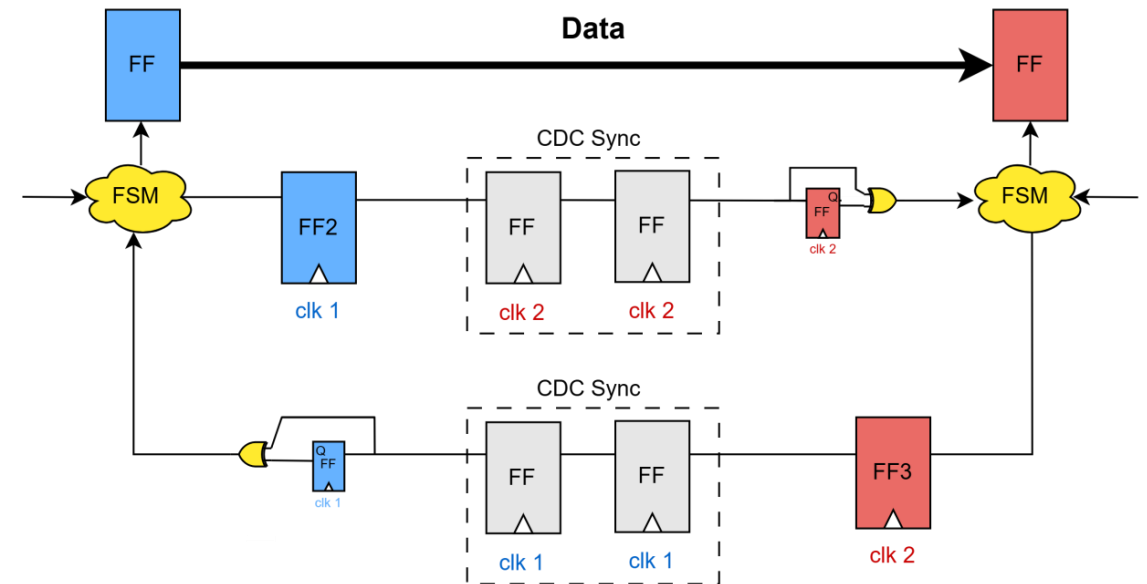
- Most designs combine the MUX synchronizer scheme with a handshake loop, where the sending domain keeps holding the data until the receiving domain responds with an acknowledgment signal that it has successfully captured the data.
- This has several benefits over the previous approach:**
  - It ensures reliable data transfer. We only change the data when we receive a confirmation it was safely captured. So no need for the previous analysis
  - It allows the receiving domain to delay the transfer if it's not ready to accept it.
- The operation goes as follows<sup>1</sup>:**
  - Domain 1 sends a request/enable signal to domain 2 through FF syncs and at the same time it launches the data to domain 2.
  - Domain 1 will keep holding the data and request signals until it receives an acknowledgment
  - After a while, domain 2 receives the request. If it's ready to accept it, it will capture the data sent from domain 1 and send an acknowledgment back through FF syncs. If it's not ready it will keep the ack signal low.
  - After a while, domain 1 receives the acknowledgment and lowers the request signal



[1]: You can watch an animation of this scheme here : <https://lnkd.in/e3mb4D57>

# Mux Synchronizers – Disadvantage

- The main disadvantage of the MUX synchronizer scheme is that it doesn't allow domain 1 to send data every clock cycle
- Instead, we hold the data stable for some time waiting for it to be captured and also waiting for an ack signal.
- This latency makes MUX schemes not suitable for designs that require high-speed processing.
- In the next part we will study another CDC method that can handle high-speed requirements.



# References

---


- 1) [http://www.sunburst-design.com/papers/CummingsSNUG2008Boston\\_CDC.pdf](http://www.sunburst-design.com/papers/CummingsSNUG2008Boston_CDC.pdf)
- 2) <https://www.uio.no/studier/emner/matnat/ifi/IN3160/v21/timeplan/in3160-l92-clock-domains.pdf>
- 3) <https://ieeexplore.ieee.org/document/1676187>
- 4) <https://www.edn.com/keep-metastability-from-killing-your-digital-design/>
- 5) <https://people.ece.ubc.ca/~edc/7660.jan2018/lec11.pdf>
- 6) <https://www.onsemi.com/pub/Collateral/AN1504-D.PDF>

# Clock Domain Crossing

Part 6 – CDC FIFO

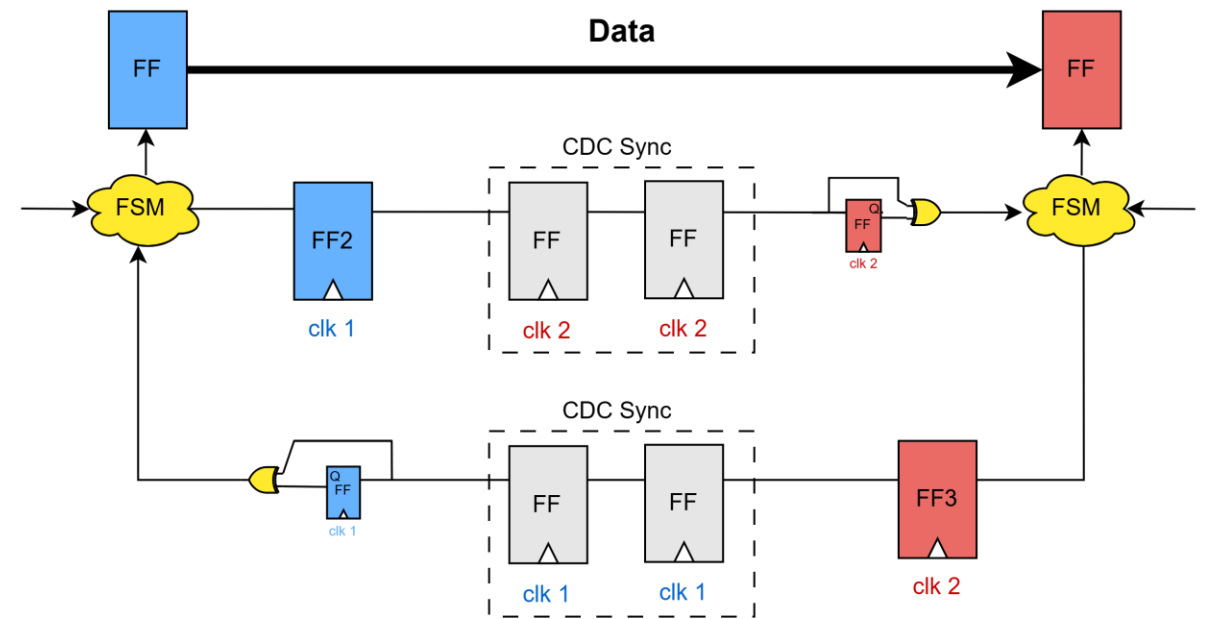
---

Amr Adel Mohammady

 /amradelm

# Introduction

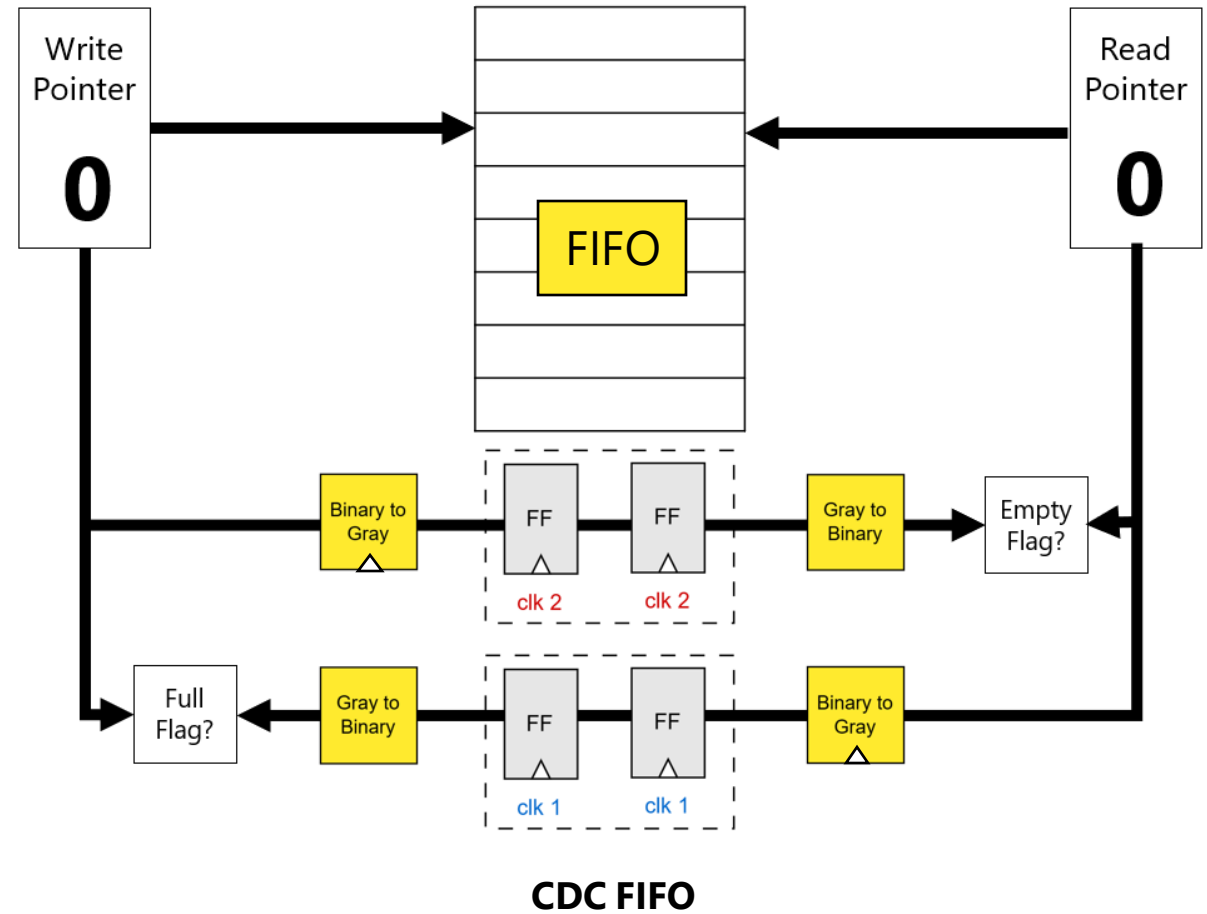
- In the previous parts we saw how to send multi-bit data signal using the mux synchronization scheme.
- Our issue with that scheme is that it's slow since it requires a handshake
- **Our CDC concerns till now:**
  - Data corruption: **Partially fixed**. The system can send multi-bit control signals and data but in a slow manner
  - Data incoherence: **Fixed**. The metastable value settles within the synchronizers at 0 or 1 and then propagates to all the domain 2 blocks with the same settled value
  - Data loss: **Fixed**. The pulse is wide enough that it won't be missed by domain 2
  - Data duplication: **Fixed** if we use pulse/edge synchronizers
  - Chip burning: **Fixed**. We limited the metastability propagation between the synchronizers and reduced its occurrence frequency.
- In this part we will see how to send multi-bit data signal using the CDC FIFO scheme



**CDC Handshake Protocol**

# Introduction

- The asynchronous FIFO is perhaps the most common way to send data across clock domains.
- **The asynchronous FIFO consists of:**
  - A circular FIFO
  - Binary to Gray encoder and Gray to binary decoder.
  - Flip-Flop synchronizers
  - Comparators
- In the next slides we will talk about each part in detail.



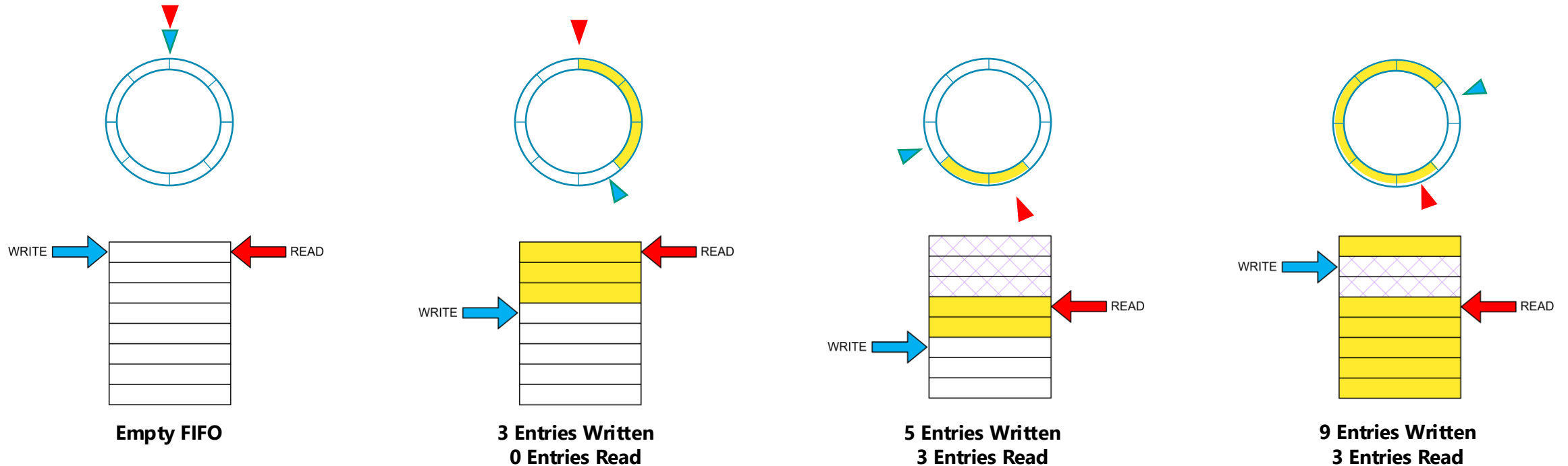
---

# First-In First-Out Buffer (FIFO)



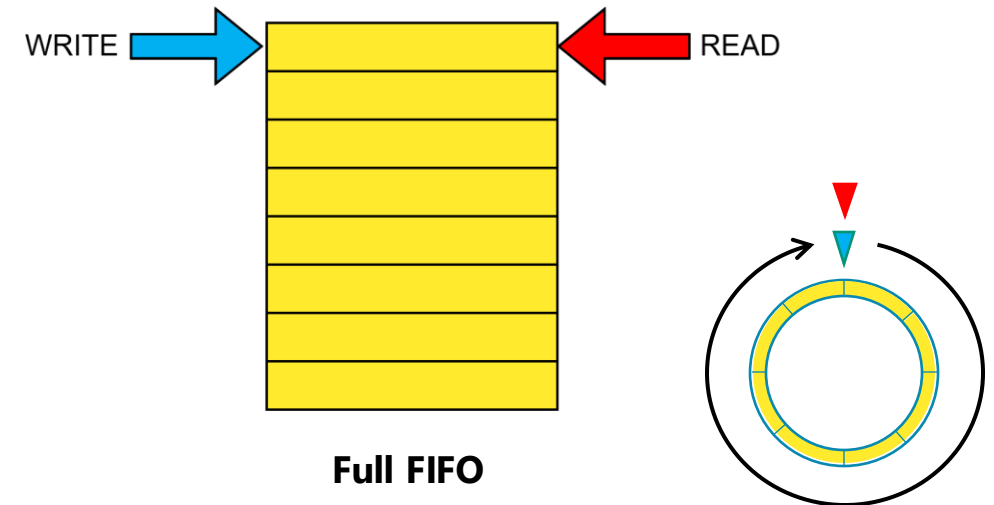
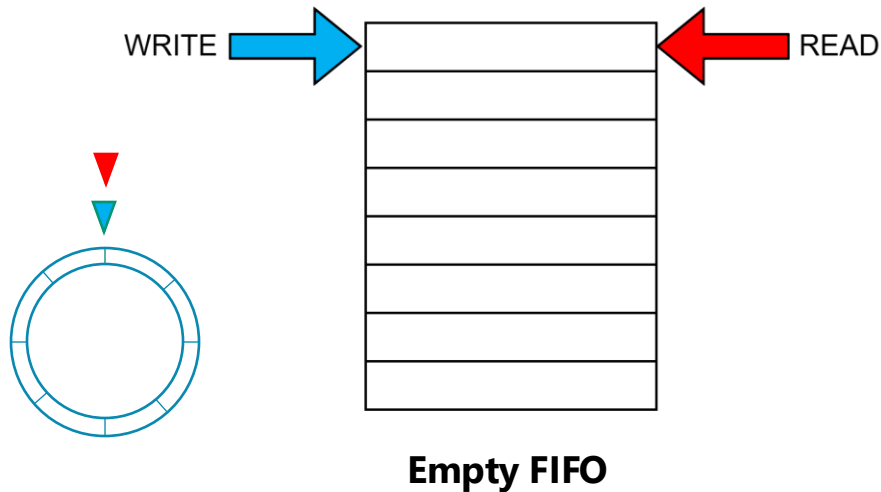
# FIFO Structure

- Circular FIFO is implemented using a fixed-size buffer (array) with two pointers: one for the read position (read pointer) and one for the write position (write pointer).
- When the write pointer reaches the end of the buffer, it wraps around to the beginning of the buffer, continuing to write data in a circular fashion. The same applies to the read pointer when reading data.
- Circular FIFO follows the First-In-First-Out principle, meaning that the first data written into the buffer is the first data read out.
- You can watch an animation of the FIFO operation here : <https://lnkd.in/ezJMAtbG>




# Empty And Full Conditions

- Underflow occurs when the buffer is empty, and a read operation is attempted. This can be managed by stopping reads when the buffer is empty.
- Overflow occurs when the buffer is full, and new data tries to overwrite unread data. This can be managed by stopping writes when the buffer is full.
- The question now is: how to determine if the FIFO is empty or Full?
- **From the 2 diagrams below we can see that when the two pointers are equal if the FIFO is either empty or full but we can't tell which.**
  - The full condition happens when the write pointer wraps around and is a complete cycle ahead of the read pointer.
  - If there is a flag that indicates wrapping around, we will be able to tell if the FIFO is empty or full
  - The conditions are therefore:
    - **Empty** : Pointers are equal and wrap flags are also equal.
    - **Full** : Pointers are equal and wrap flags are not equal.



# Empty And Full Conditions Implementation

- To implement the write and read pointer with the added flag you have to do the following when the pointer reaches the max value (the end):
  - Reset the pointer
  - Toggle the flag
- If the FIFO depth is a power of 2 you don't need to create additional logic to handle the resetting and toggling.
- That's because once the pointer reaches the max value it will overflow and thus automatically reset the pointer and toggle the most significant bit (MSB)



Flag	Ptr
0	000
0	001
0	010
0	011
0	100
1	000
1	001
1	010
1	011
1	100
0	000

5-Deep FIFO

Flag	Ptr
0	000
0	001
0	010
0	011
0	100
0	101
0	110
0	111
1	000

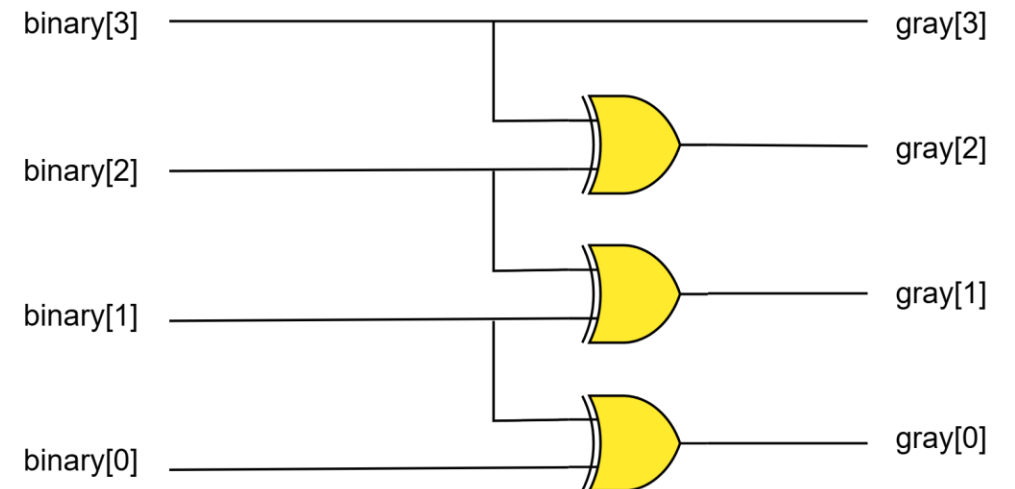
8-Deep FIFO

---

# Gray Encoder And Decoder

# Binary to Gray Encoder

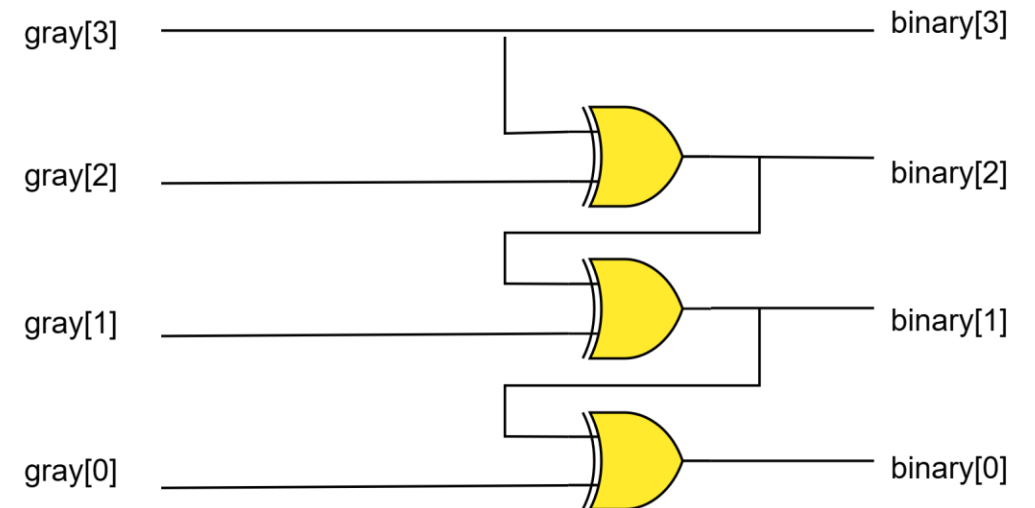
- Gray code is a binary numeral system where two successive values differ in only one bit.
- **To convert a binary number to its corresponding Gray code:**
  - The most significant bit (MSB) of the Gray code is the same as the MSB of the binary input.
  - Each subsequent Gray code bit is derived by XOR-ing the corresponding binary bit with the next higher-order binary bit.
  - Example: If binary\_in is 3'b101 (binary), then:
    - $\text{gray\_out}[2] = \text{binary\_in}[2] = 1$
    - $\text{gray\_out}[1] = \text{binary\_in}[2] \wedge \text{binary\_in}[1] = 1 \wedge 0 = 1$
    - $\text{gray\_out}[0] = \text{binary\_in}[1] \wedge \text{binary\_in}[0] = 0 \wedge 1 = 1$
    - So, gray\_out will be 3'b111 (Gray code).



**4-Bit Binary to Gray**

# Gray to Binary Decoder

- **To convert a Gray Code to its corresponding binary code:**
  - The most significant bit (MSB) of the Gray code is the same as the MSB of the binary input.
  - Each subsequent binary bit is derived by XOR-ing the previous binary bit with the corresponding Gray code bit.
  - Example: If gray\_in is 3'b111 (Gray), then:
    - $\text{binary\_out}[2] = \text{gray\_in}[2] = 1$
    - $\text{binary\_out}[1] = \text{binary\_out}[2] \oplus \text{gray\_in}[1] = 1 \oplus 1 = 0$
    - $\text{binary\_out}[0] = \text{binary\_out}[1] \oplus \text{gray\_in}[0] = 0 \oplus 1 = 1$
    - So, binary\_out will be 3'b101 (binary code).



**4-Bit Gray to Binary**

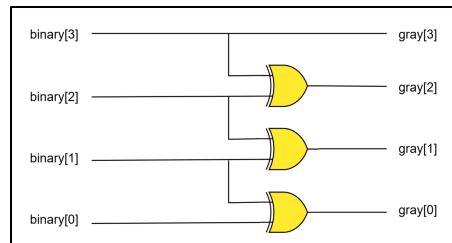
# Verilog Code

```
module binary_to_gray #(
    parameter N = 3 // Parameter to define the bit-width,
    default is 3
) (
    input  [N-1:0] binary_in, // N-bit binary input
    output [N-1:0] gray_out   // N-bit Gray code output
);

// Assigning the MSB directly as it remains the same
assign gray_out[N-1] = binary_in[N-1];

// Loop to calculate the Gray code output from binary
genvar i;
generate
    for (i = N-2; i >= 0; i = i - 1) begin : bin_to_gray
        assign gray_out[i] = binary_in[i+1] ^ binary_in[i];
    end
endgenerate

endmodule
```



**Binary to Gray**

```
module gray_to_binary #(
    parameter N = 3 // Parameter to define the bit-width,
    default is 3
) (
    input  [N-1:0] gray_in, // N-bit Gray code input
    output [N-1:0] binary_out // N-bit binary output
);

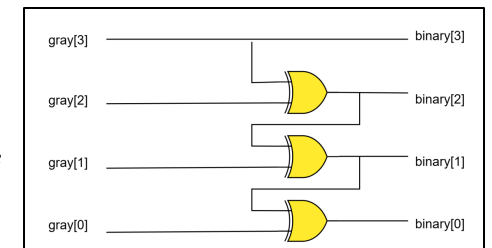
// Internal wire to store intermediate results
wire [N-1:0] binary_temp;

// Assigning the MSB directly as it remains the same
assign binary_temp[N-1] = gray_in[N-1];

// Loop to calculate the binary output from Gray code
genvar i;
generate
    for (i = N-2; i >= 0; i = i - 1) begin : gray_to_bin
        assign binary_temp[i] = binary_temp[i+1] ^
gray_in[i];
    end
endgenerate

// Assign the result to output
assign binary_out = binary_temp;

endmodule
```



**Gray to Binary**

---

# CDC FIFO



# CDC FIFO

---

- **Write Interface (Write Logic)**

- Function: The write interface is responsible for writing data into the FIFO. It operates in the write clock domain (write\_clk).
- Components:
  - Write Pointer (write\_ptr): Tracks the position in the FIFO where the next data element will be written.
  - Write Enable Signal: Controls when data can be written to the FIFO, typically asserted when the FIFO is not full.
  - Write Data (data\_in): The actual data to be written into the FIFO.
- Process: Each time a write operation occurs (triggered by the write clock), the data is stored at the location indicated by the write pointer, and the write pointer is incremented.

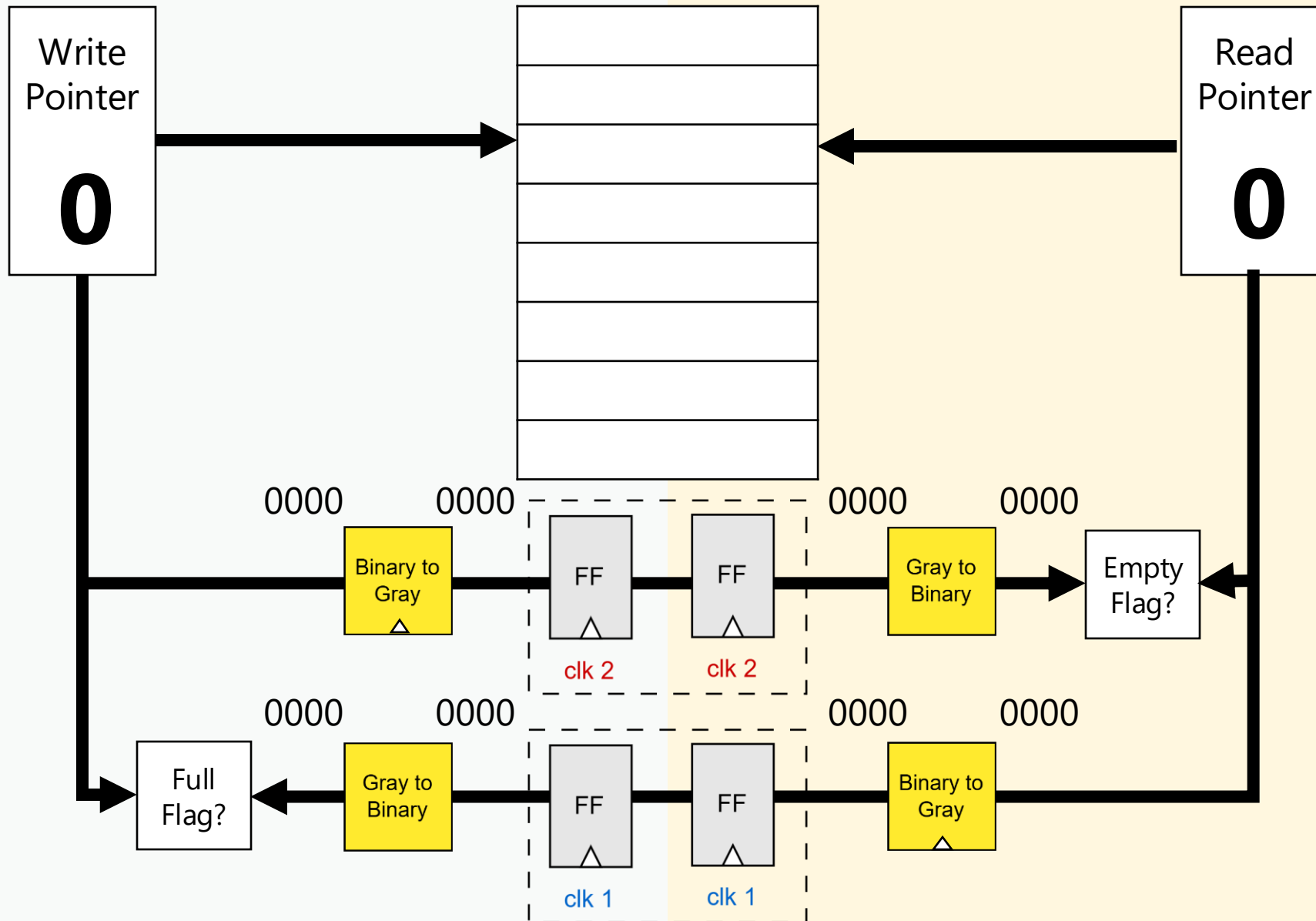
- **Read Interface (Read Logic)**

- Function: The read interface is responsible for reading data out of the FIFO. It operates in the read clock domain (read\_clk).
- Components:
  - Read Pointer (read\_ptr): Tracks the position in the FIFO from where the next data element will be read.
  - Read Enable Signal: Controls when data can be read from the FIFO, typically asserted when the FIFO is not empty.
  - Read Data (data\_out): The data read from the FIFO, presented to the output.
- Process: Each time a read operation occurs (triggered by the read clock), the data is read from the location indicated by the read pointer, and the read pointer is incremented.

- An animation of the operation can be found here : <https://lnkd.in/eZUCAx9a>

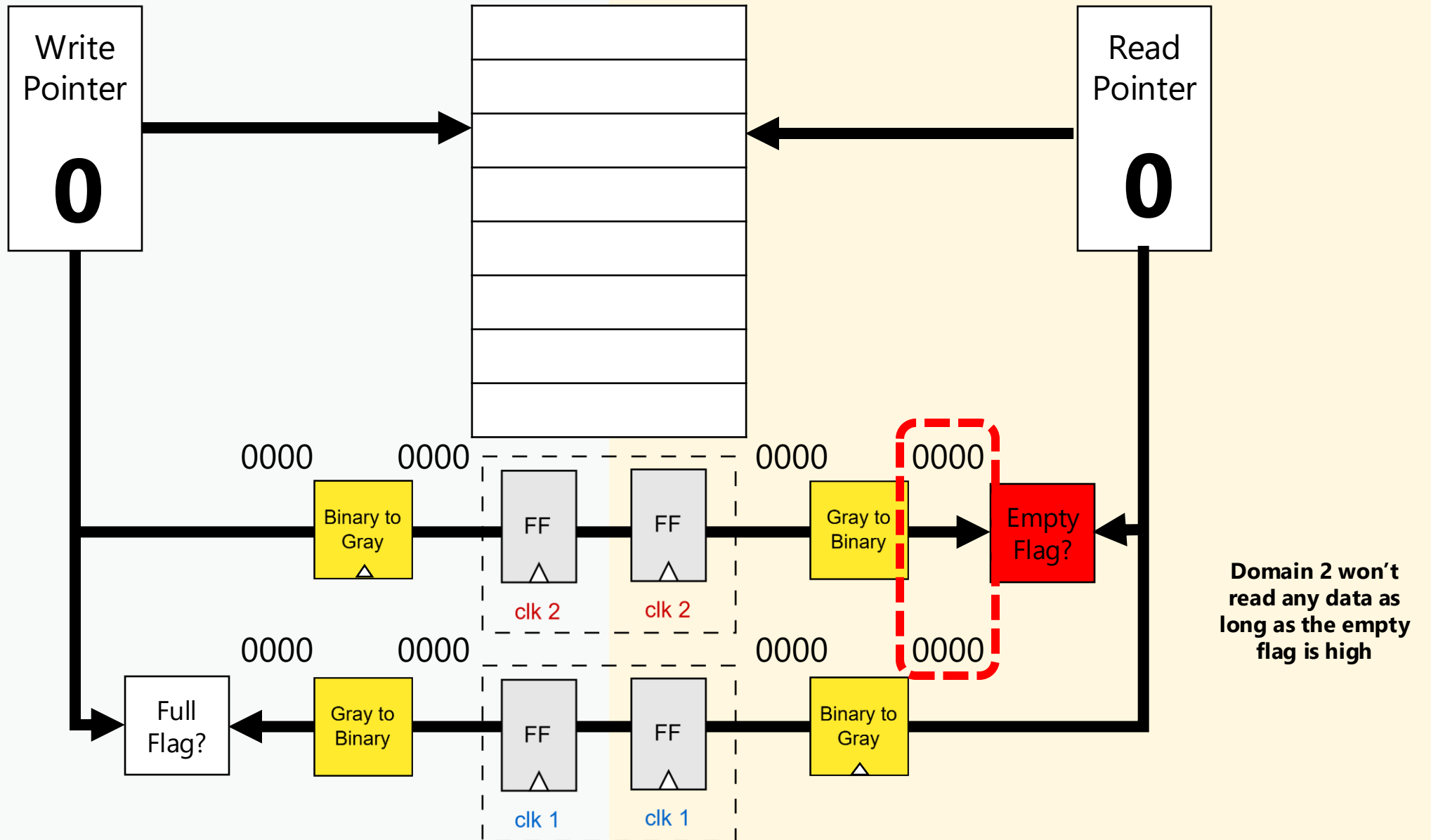
## Domain 1

## Domain 2



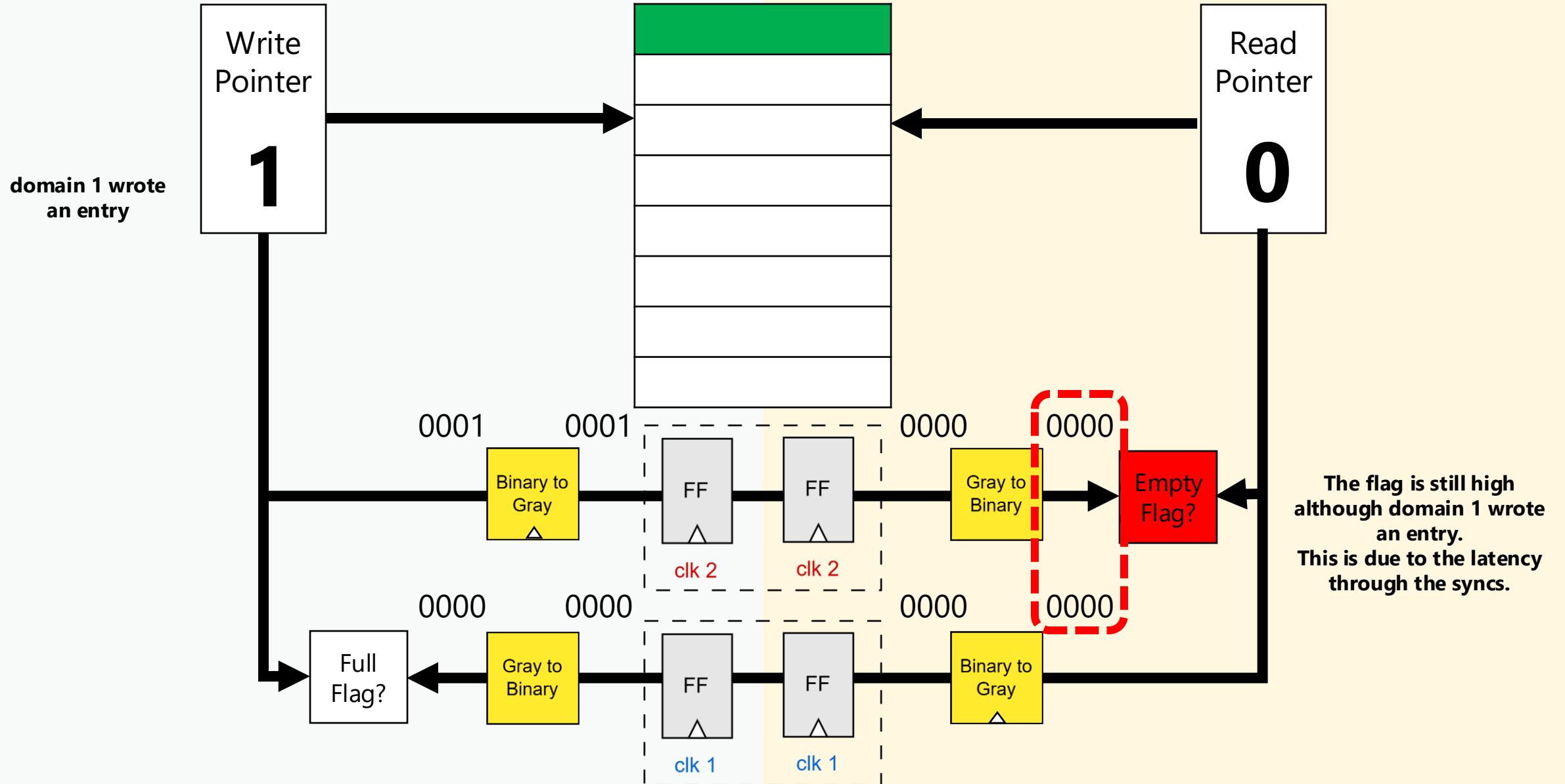
## Domain 1

## Domain 2



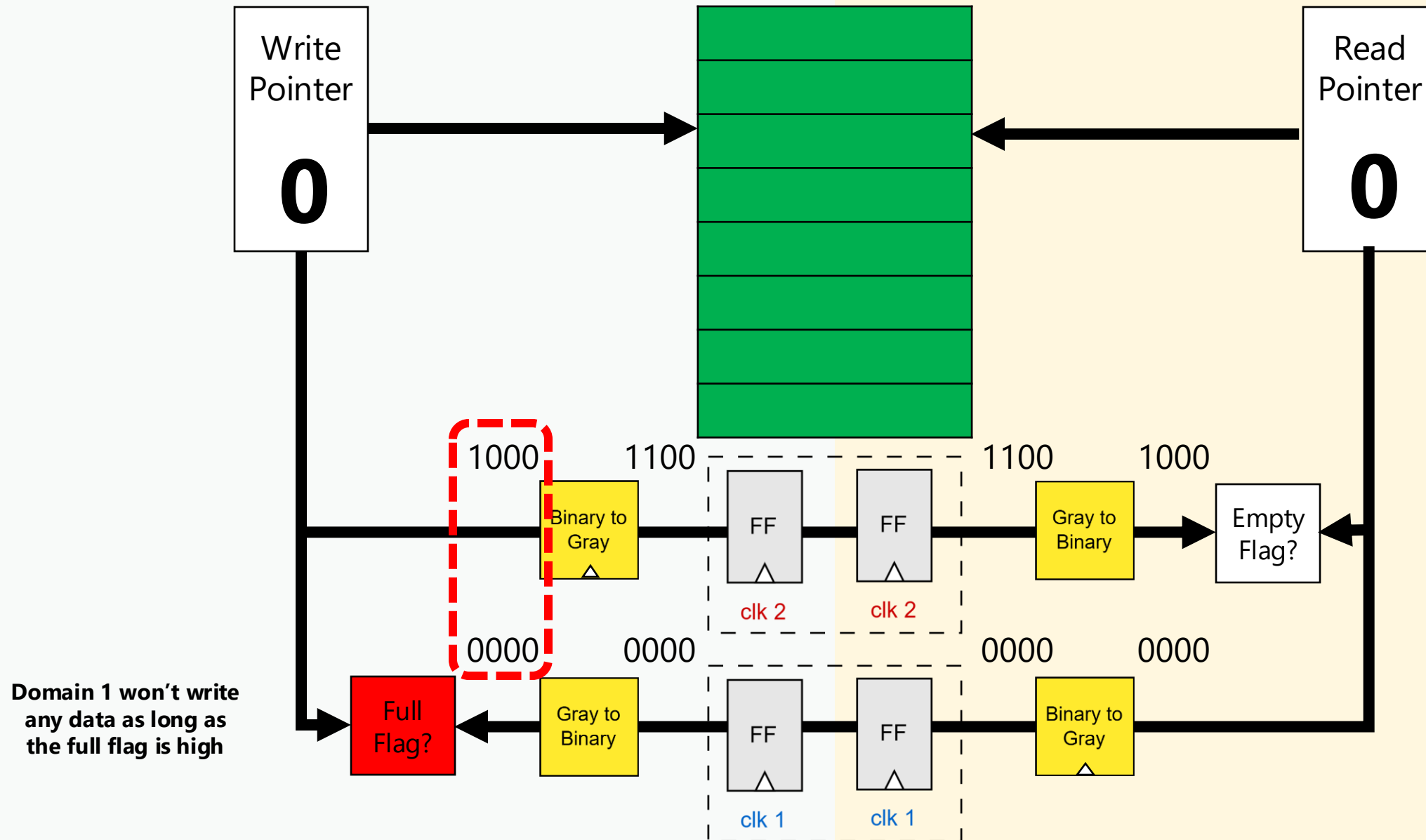
## Domain 1

## Domain 2



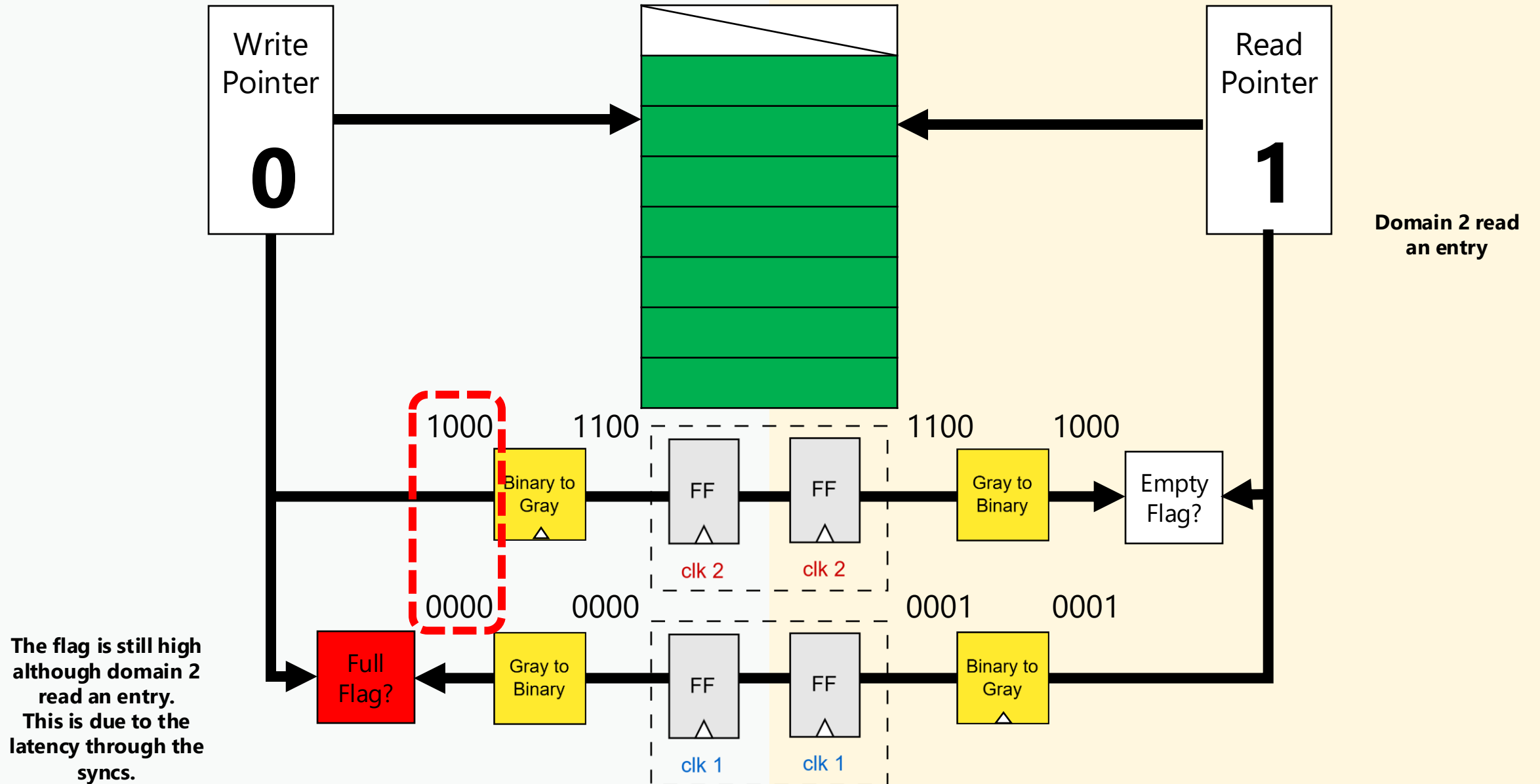
## Domain 1

## Domain 2



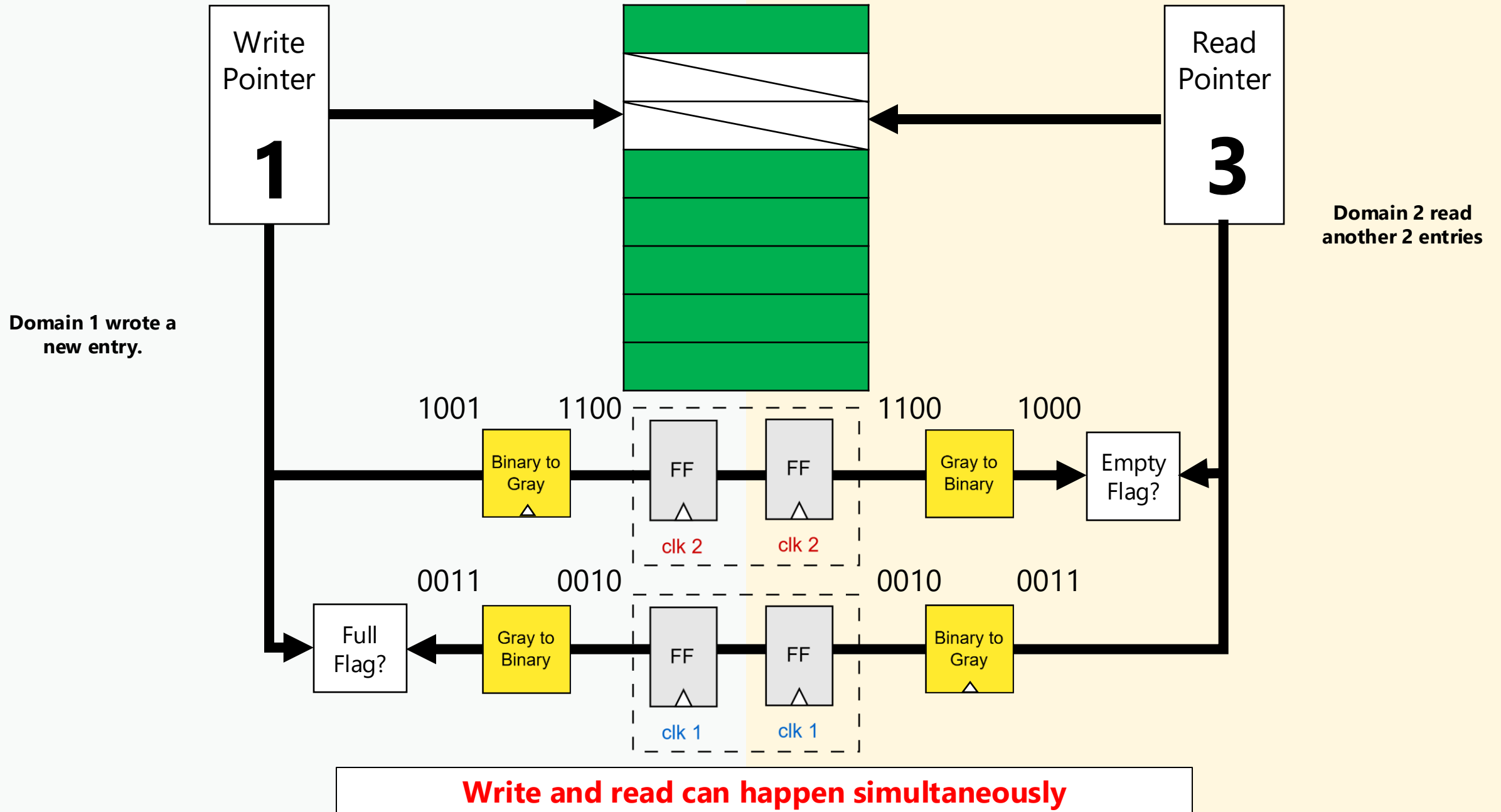
## Domain 1

## Domain 2



## Domain 1

## Domain 2



# FIFO Depth

---

- FIFO depth is crucial to prevent data overflow (when the FIFO is too shallow) or underutilization (when the FIFO is too deep).
- The data arrives in bursts (periods of high activity followed by inactivity), the FIFO must be deep enough to hold all the burst data until it can be processed.
- If the write domain writes **N** data samples within time **T** and the read domain reads **M** data samples within the same time we end up with **N-M** samples that need to be stored in the FIFO.
- For example if the write domain is 3 times faster than the read domain. For every 3 samples written the slow domain will be able to read only one sample.
- If the write domain write a burst of N samples. The read domain will only read  $\frac{N}{3}$  samples leaving  $\frac{2N}{3}$  samples that need to be stored
- **The size is therefore calculated as**

$$Depth = Burst - Burst \times \frac{f_{read}}{f_{write}}$$

$$Depth = Burst \times \left(1 - \frac{f_{read}}{f_{write}}\right)$$



# FIFO Depth - Examples

---

- Let  $f_{write} = 400 \text{ MHz}$ ,  $f_{read} = 150 \text{ MHz}$ . Burst size = 120. There are no idle cycles between consecutive writes or read operations.
  - $Depth = Burst \left(1 - \frac{f_{read}}{f_{write}}\right) = 120 \left(1 - \frac{150}{400}\right) = 75 \text{ location.}$
  - The size rounded to power of 2 = 128 locations.
- Let  $f_{write} = 400 \text{ MHz}$ ,  $f_{read} = 150 \text{ MHz}$ . Burst size = 120. There is 1 idle cycle between consecutive writes.
  - The idle cycle will slow down the writing operation giving the read operation more time to read. We should expect the required size to decrease
  - With the idle cycle, the write domain takes 2 cycles for each write instead of 1. Effectively the frequency is halved.
  - $Depth = Burst \left(1 - \frac{f_{read}}{f_{write}/2}\right) = 120 \left(1 - \frac{150}{400/2}\right) = 30 \text{ location.}$
  - The size rounded to power of 2 = 32 locations.
- More examples can be found in this document by **Putta Satish** : <https://shorturl.at/qBrGl>

# References

---

- 1) [http://www.sunburst-design.com/papers/CummingsSNUG2008Boston\\_CDC.pdf](http://www.sunburst-design.com/papers/CummingsSNUG2008Boston_CDC.pdf)
- 2) <https://www.uio.no/studier/emner/matnat/ifi/IN3160/v21/timeplan/in3160-l92-clock-domains.pdf>
- 3) <https://ieeexplore.ieee.org/document/1676187>
- 4) <https://www.edn.com/keep-metastability-from-killing-your-digital-design/>
- 5) <https://people.ece.ubc.ca/~edc/7660.jan2018/lec11.pdf>
- 6) <https://www.onsemi.com/pub/Collateral/AN1504-D.PDF>

# Clock Domain Crossing

Part 7 – Timing Constraints

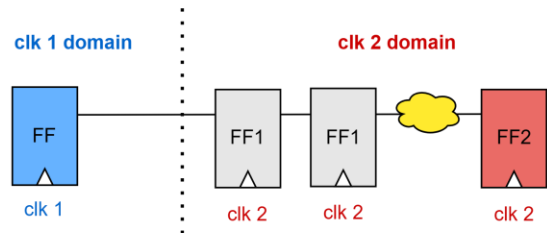
---

Amr Adel Mohammady

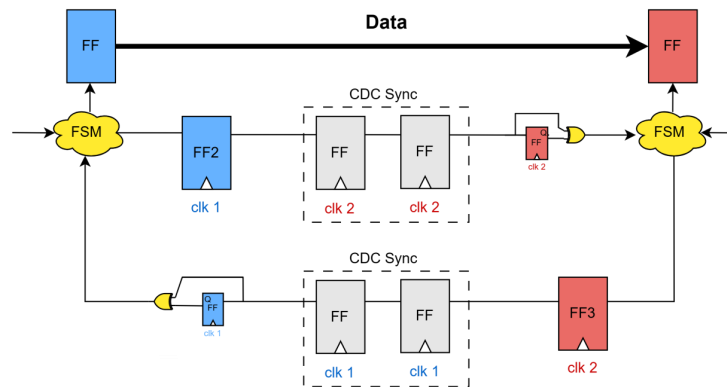
 /amradelm

# Introduction

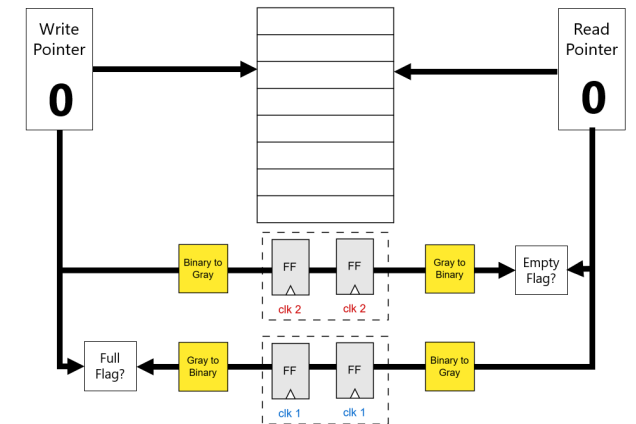
- In the previous parts we went through all the CDC solutions and schemes.
- In this part we will discuss the timing constraints associated with these schemes.
- We mentioned that CDC paths are asynchronous and therefore can't be analyzed with static timing analysis. '
- That's why, in the past, the most common approach was applying false paths on CDC paths. We will see how that may lead to major issues.
- It turns out we still need some timing constraints to enforce some assumptions we made when designing the CDC circuit



**CDC Synchronizers**



**CDC Handshake Protocol**



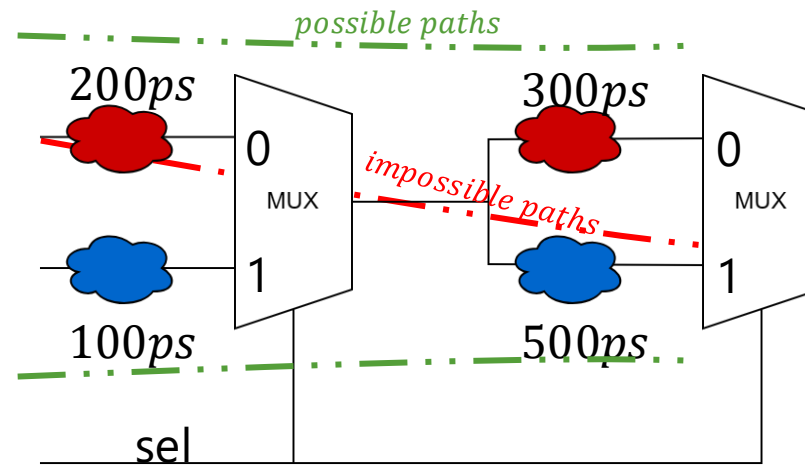
**CDC FIFO**

---

1<sup>st</sup> (Trial) Solution :  
Apply False Path

# What is a False Path?

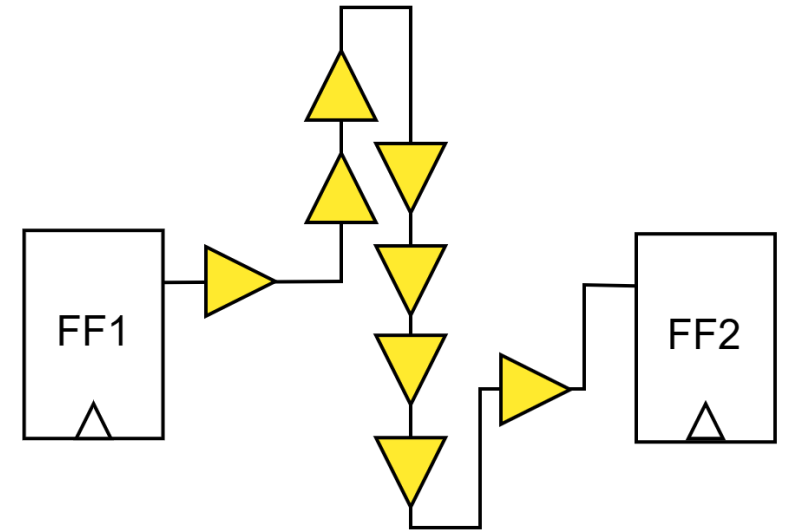
- False paths are timing paths that can't possibly occur due to the logic of the circuit
- Consider the example below:**
  - Both muxes have the same select signal. This means we have 2 possible timing paths. The one going through both **red** logics ( $200 + 300 = 500ps$ ) and the one going through both **blue** logics ( $100 + 500 = 600ps$ )
  - The paths going through a **red** logic then a **blue** logic ( $200 + 500 = 700ps$ ) or **blue** logic then **red** logic ( $100 + 300 = 400ps$ ) is impossible to happen.
  - Unless we instruct the tool to ignore these false paths, they will be considered for timing analysis leading to the large  $T_{comb}$  of the red to blue path which will violate setup.



# What Would Happen With A False Path Constraint?

---

- Applying false path will make the tool ignore the timing of the paths and therefore may create unnecessary delay that breaks our CDC circuit.
  - The tool might:
    - Place the launch and capture FF far apart.
    - Create unnecessarily long routes.
    - Add unnecessary buffers in the routes.
    - Use slow cells/FFs to save power.
  - We will see how this may break our CDC circuits. We will consider 2 examples
    - CDC Mux
    - CDC Gray Coding

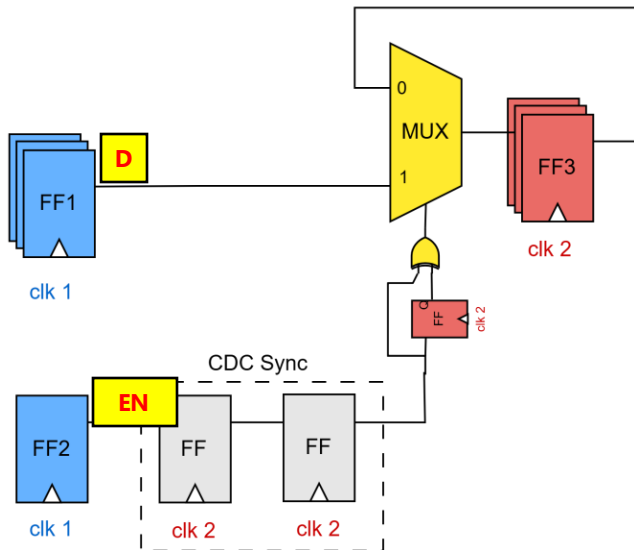


**Possible Schematic of a very relaxed Path**

# CDC Mux Scheme And False Path

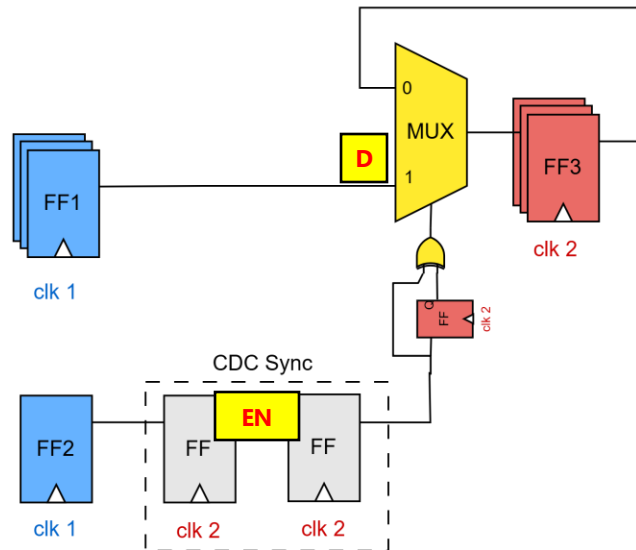
## 1 Let's review the CDC MUX scheme<sup>1</sup>:

- The data goes directly to the Rx domain MUX
- The enable goes to the Rx domain through FF synchronizers



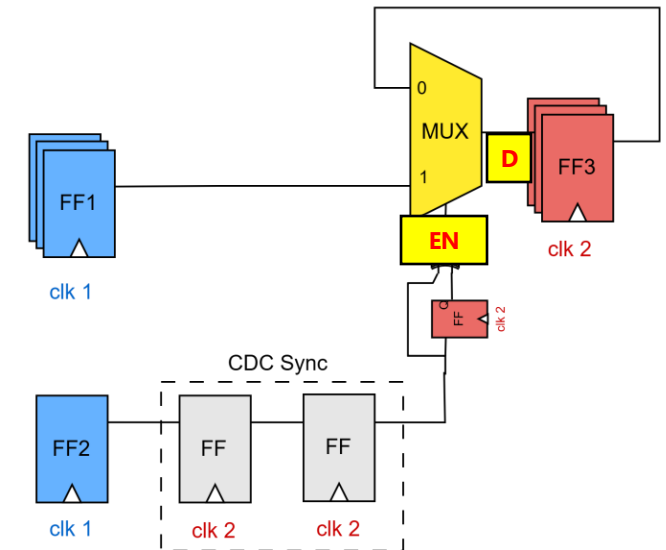
## 2

- The data reaches the MUX quickly since no FF exist in the way.
- The enable is still going through the FF syncs



## 3

- After some time, the enable arrives at the MUX and opens the gate for the data.
- Since the enable is synchronized, it's guaranteed with STA that the gate will open without violating setup or hold time and therefore the data won't cause metastability



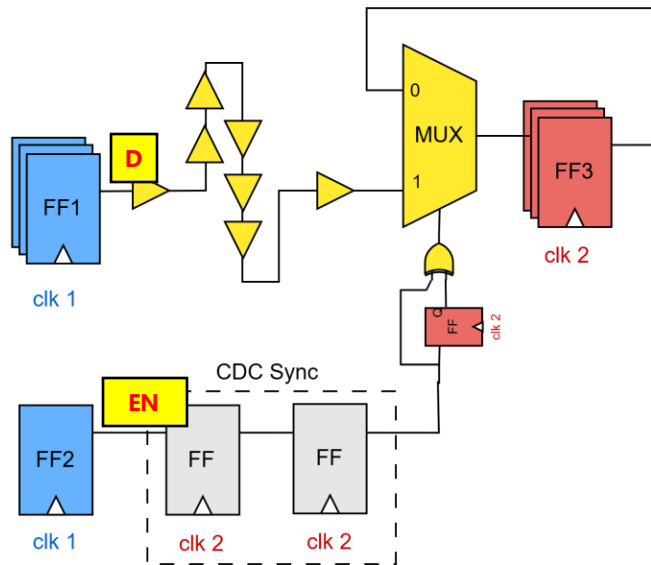
[1]: You can watch an animation of this here: <https://lnkd.in/en-iuNPx>



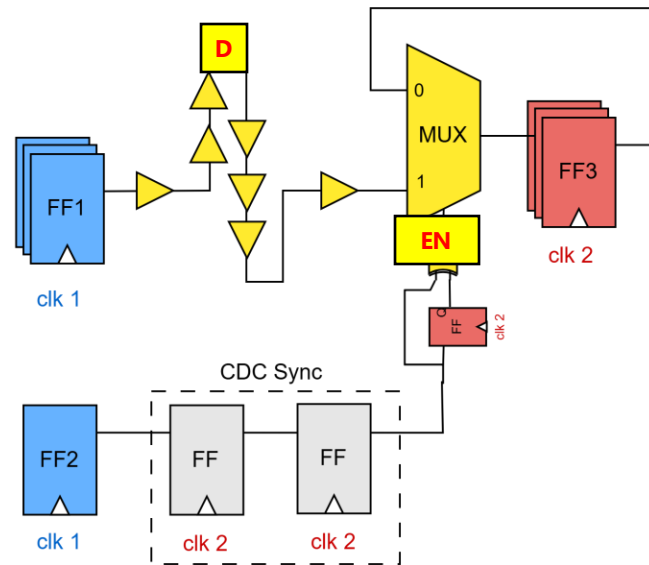
# CDC Mux Scheme And False Path

## 1 Now let's consider the faulty behavior with a false path constraint applied<sup>1</sup>

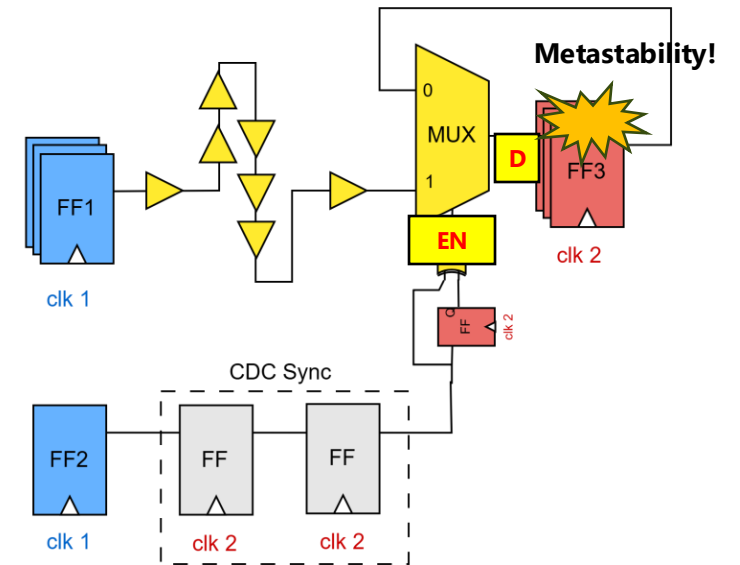
- The data goes to the Rx domain MUX
- The enable goes to the Rx domain through FF synchronizers



- ## 2
- The data takes a long time to reach the MUX due to the logic delay.
  - The enable reaches the MUX and opens the gate



- ## 3
- After some time, the data arrives at the MUX after the enable.
  - The data is a domain 1 signal arriving at domain 2 FF. Metastability happens

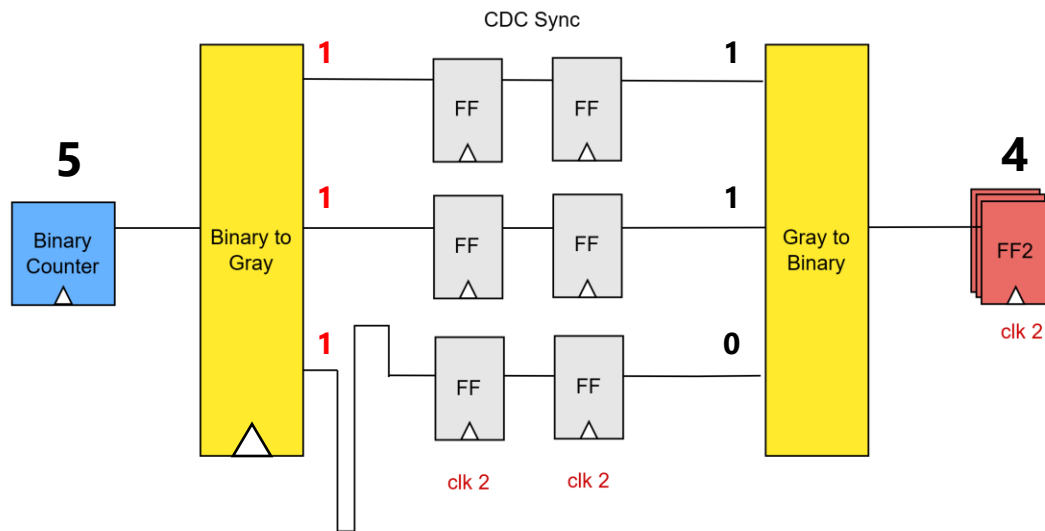


[1]: You can watch an animation of this here: <https://lnkd.in/en-iuNPx>

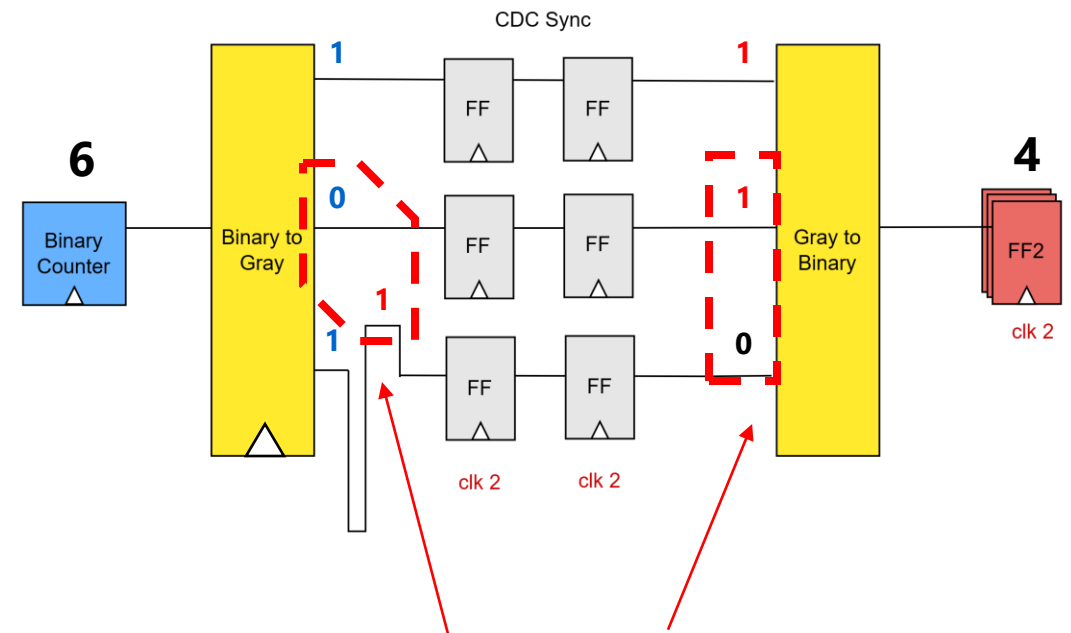
# CDC Gray Coding And False Path

## 1 Now let's see the faulty behavior with CDC Gray coding

- Initially the Rx sees gray code 110 (decimal 4)
- After that we send gray code 111 (decimal 5)



- ## 2
- We then send gray code 101 (decimal 6)
  - Due to the long delay on the LSB line, the logic "1" from decimal 5 didn't reach the Rx yet.
  - Now the Rx sees two bit changing and may jump to wrong count



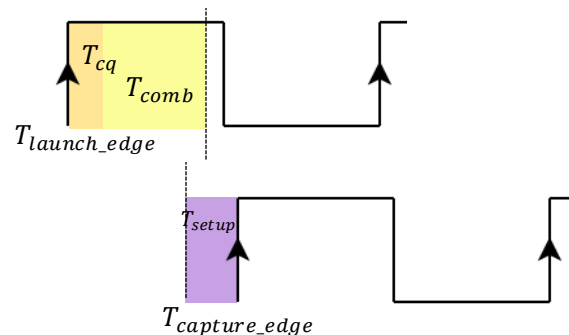
Multiple bits are changing at the same time.  
**The Gray code is violated**

---

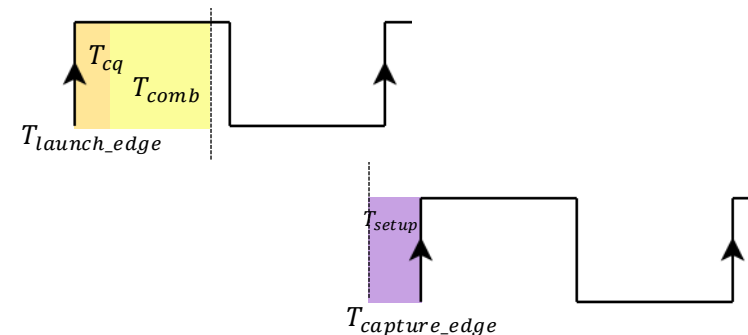
2<sup>nd</sup> (Trial) Solution :  
Don't Apply False Path

# What Could Happen Without a False Path Constraints?

- By default, the tool will assume the 2 CDC clocks are synchronous and will run STA on any path between them.
- **This will lead to one of 2 issues:**
  - If the clock skew between the 2 clocks is small, the path will be very tight and won't meet timing. The synthesis and PnR tools will spend a lot of effort trying to fix the path<sup>1</sup>.
  - If the clock skew between the 2 clocks is large, the path will be relaxed and may meet timing with a large setup margin.
    - The tool might add delay (for example, to save power)
    - We get the same issue of applying a false path constraint



**Small Skew Case**



**Large Skew Case**

[1]: The tools ignore less critical paths and focus on the critical ones. This will lead to real paths being masked by fake CDC violations

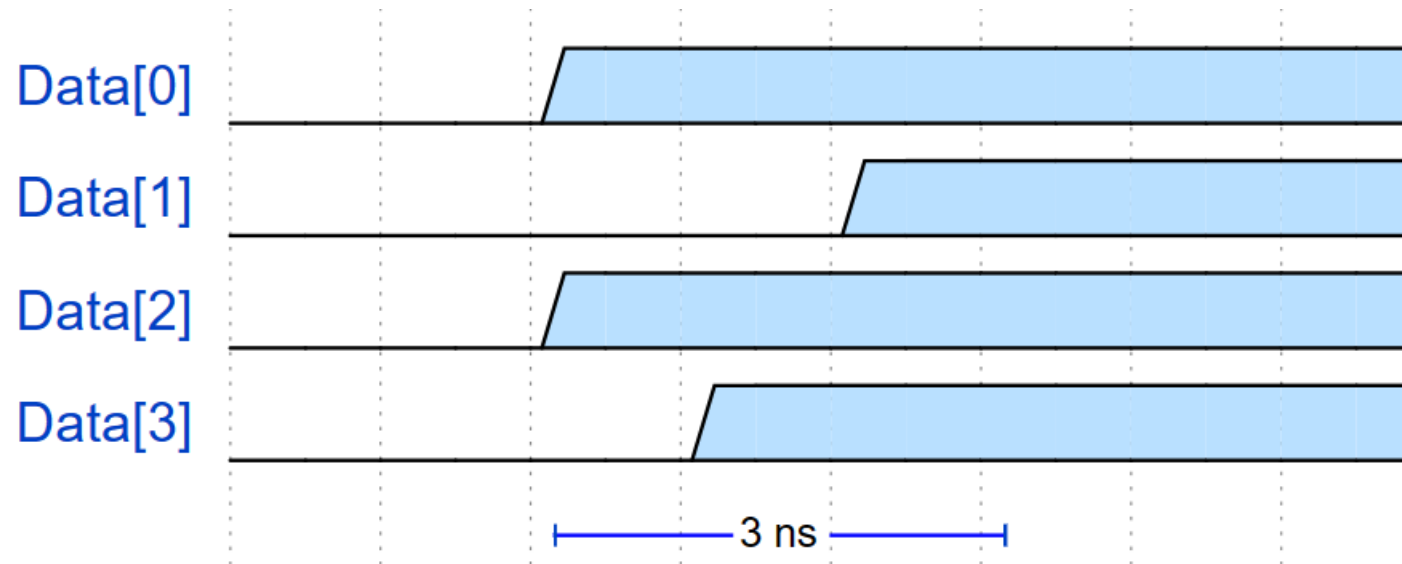
---

# 3<sup>rd</sup> (Correct) Solution : Skew Constraint

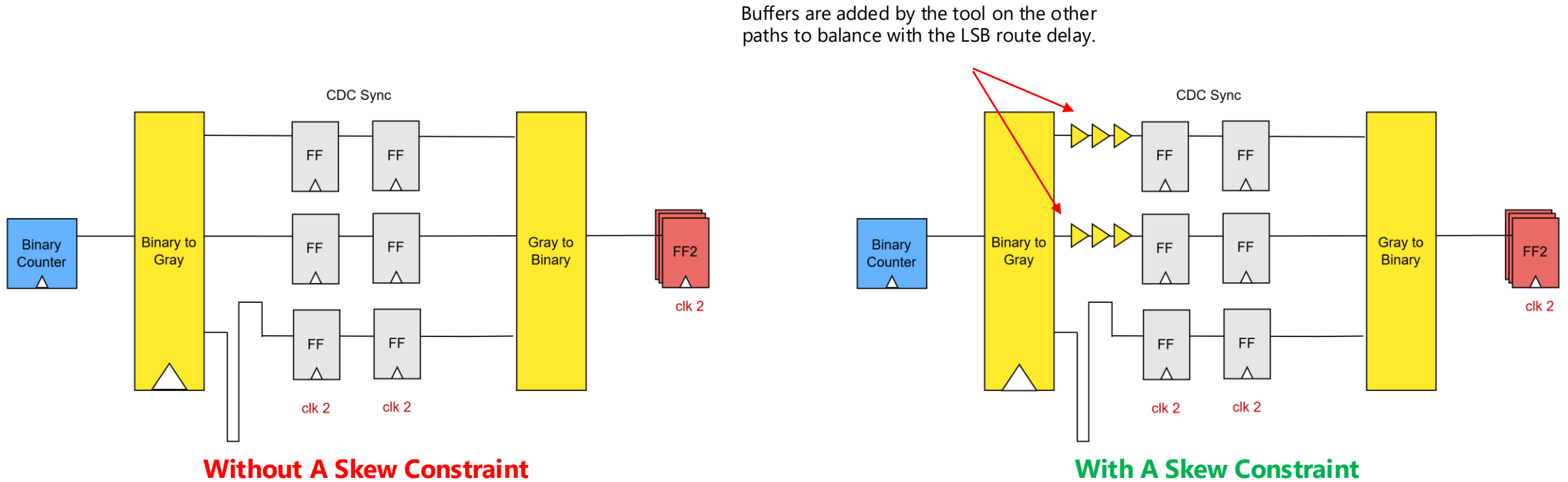
# What is A Skew Constraint

---

- Skew checks constraint the arrival difference between 2 signals or more.
- In the example below we have a data bus of 4 bits. The bits should arrive close to each other with a difference no more than 3ns. This means the difference between the latest bit to arrive and the earliest bit to arrive shouldn't exceed 3ns.
- To fix skew violations we need to speed up slow signals and/or slow down fast ones.



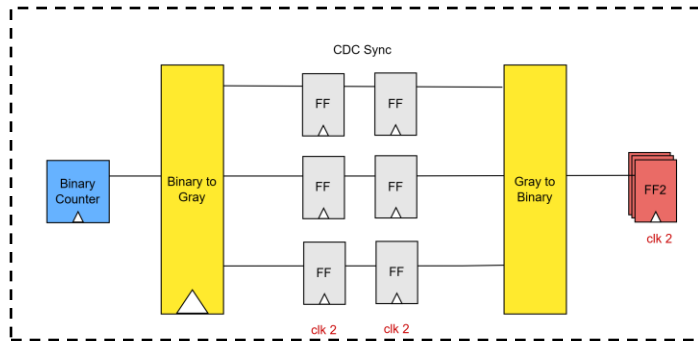
# CDC Gray Coding And Skew Constraint



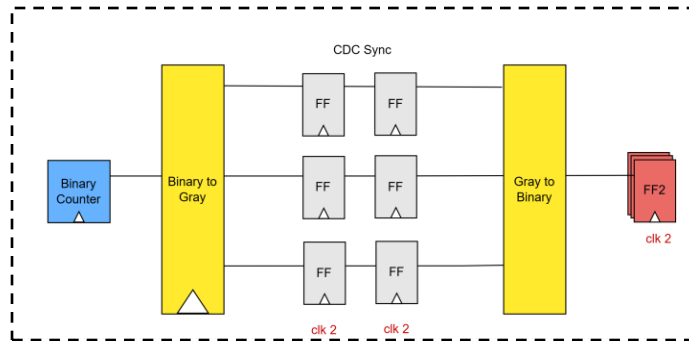
# The Issue With This Approach

- The main issue with this approach is that it needs lots of manual efforts
- If we have multiple CDC paths, we need to identify each group of signals and add skew constraints for them
- We will try another easier approach

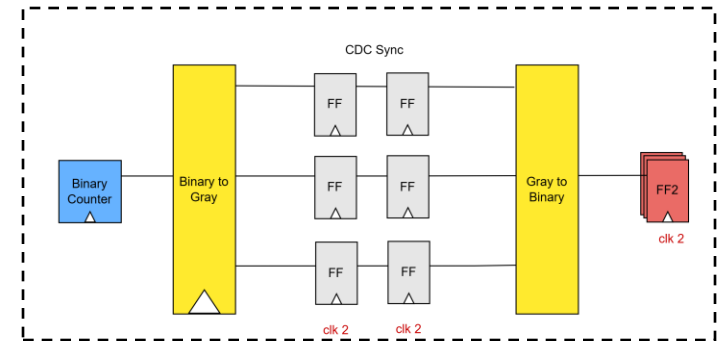
**Group 1**



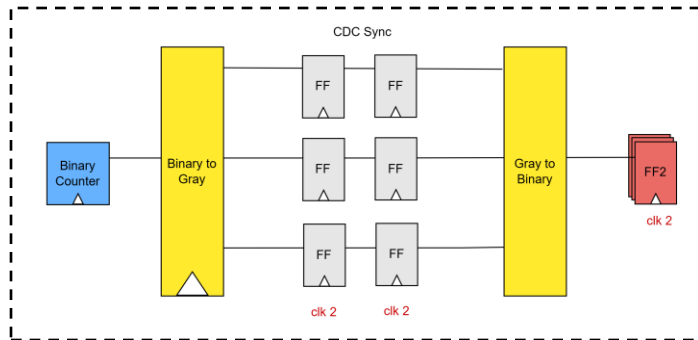
**Group 2**



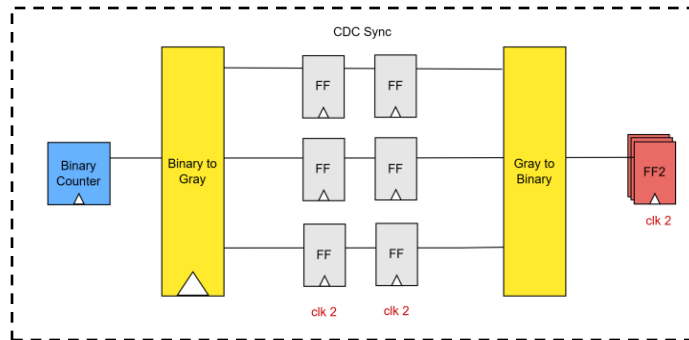
**Group 3**



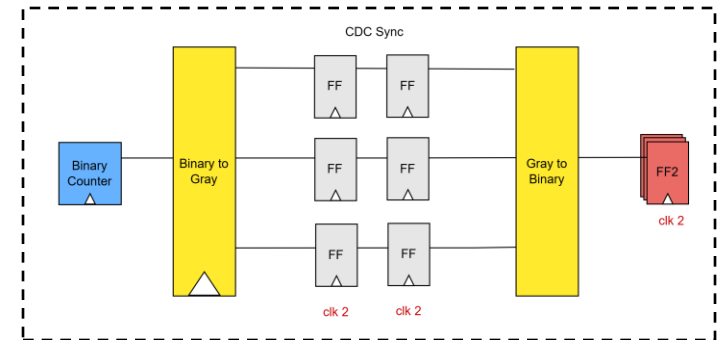
**Group 4**



**Group 5**



**Group 6**



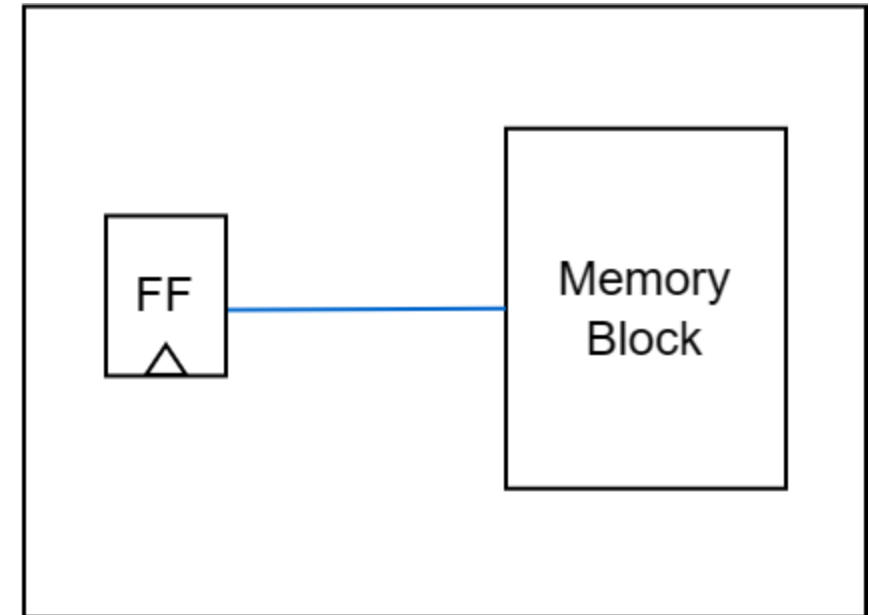
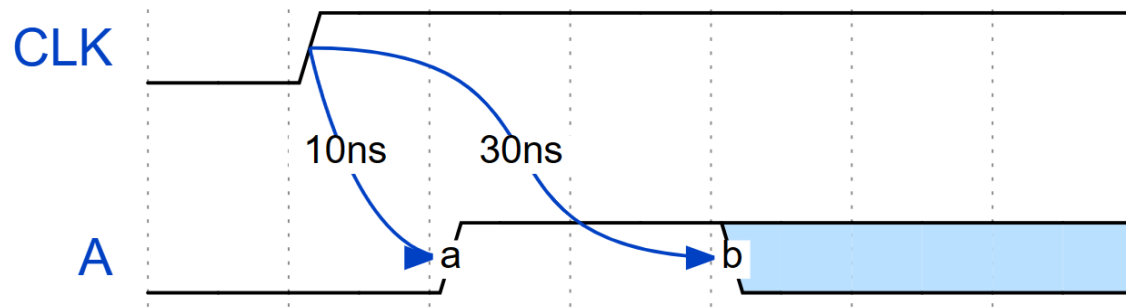


---

4<sup>th</sup> (Best) Solution :  
Max Delay Constraint

# Max and Min Delays

- Sometimes we want to control the arrival time of a signal.
- In the example below, it's required that signal A arrives at the memory block no earlier than 10ns and no later than 30ns after the clock edge.
- To constraint signal A to follow this requirement we need to apply a min delay constraint of 10ns and a max delay of 30ns<sup>1</sup>.

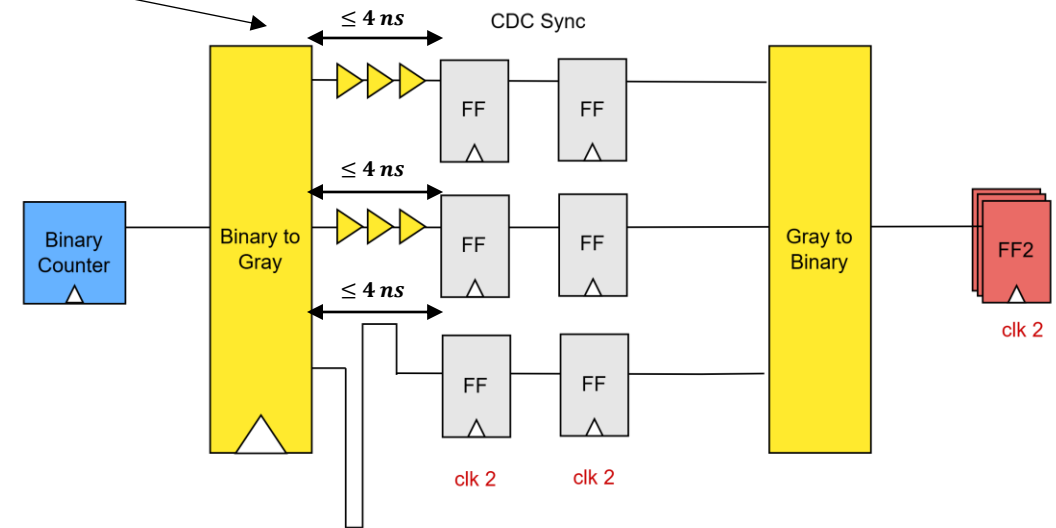


More details : <https://docs.amd.com/r/2021.2-English/ug903-vivado-using-constraints/Min/Max-Delays>

[1] : Don't apply the constraint from the Q pin of the FF but from the CK pin. Otherwise, the setup and hold timing paths of the FF will be broken

# Max Delay Constraint

- The best approach is to add a max delay constraint with an amount small enough that the CDC paths are not broken
- What makes this approach easy is that it can be applied to all CDC paths with one line<sup>1</sup> so it doesn't need manual work:
  - `pt_shell> set_max_delay 4.0 -from CLK1 -to CLK2`
- **What remains now is what value to use for the max delay constraint.**
  - In some cases, we need to apply the Tx clock period
  - In other cases, we need to apply the Rx clock period
  - In other cases, we need to apply multiple clock periods.
  - We will use the worst case (smallest) instead of applying a specific max delay value for each
- The value used for max delay might be too tight for some CDC paths. In that case, we can resort to skew constraint



[1] : In some tools, the max delay constraint overwrites the setup constraints, but in others, it won't. Depending on your tool, you might need to first apply a false path on the setup constraint then apply the max delay constraint : [Timing constraints for clock-domain crossings. #sta #cdc \(github.com\)](#)

# References

---

- 1) <https://gist.github.com/brabect1/7695ead3d79be47576890bbcd61fe426>
- 2) Y. Mirsky, O. Tsarfaty, D. Stein, & O. Winner, "Timing Analysis of Unconstrained Clock Domain Crossings – the Need and the Method,"
- 3) O. Dasa, Y. Mirsky "A New Approach to Easily Resolve the Hidden Timing Dangers of False Path Constraints on Clock Domain Crossings"

---

# Thank You!